Machine Learning Engineering for Production (MLOps) Specialization

by DeepLearning.AI

Course Site

Made By: Matias Borghi

Table of Contents

Course #1 Introduction to Machine Learning in Production 3
- **Week 1: Overview of the ML Lifecycle and Deployment 5**
  - The Machine Learning Project Lifecycle 5
  - Deployment 8
  - Week 1 Optional References 16
  - Week 1: Overview of the ML Lifecycle and Deployment 16
- **Week 2: Select and Train a Model 17**
  - Selecting and Training a Model 17
  - Why low average error isn't good enough 19
  - Error analysis and performance auditing 23
  - Performance auditing 28
  - Data iteration 29
  - A useful picture of data augmentation 30
  - Week 2 Optional References 35
  - Week 2: Select and Train Model 35
- **Week 3: Data Definition and Baseline 36**
  - Define Data and Establish Baseline 36
    - Why is data definition hard? 36
    - More label ambiguity examples 37
    - Major types of data problems 39
    - Small data and label consistency 41
    - Improving label consistency 43
    - Human level performance (HLP) 44
    - Raising HLP 46
  - Label and Organize Data 47
    - Obtaining data 47
    - Data pipeline 49
    - Meta-data, data provenance and lineage 50
    - Balanced train/dev/test splits 52
  - Scoping (optional) 52
    - What is scoping? 52
    - Scoping process 53
    - Diligence on feasibility and value 54
    - Diligence on value 56
    - Milestones and resourcing 57
  - Week 3 Optional References 59
  - Week 3: Data Definition and Baseline 59
- References 60

Course #2 Machine Learning Data Lifecycle in Production 61
- **Week 1: Collecting, Labeling and Validating Data 63**

Introduction to Machine Learning Engineering in Production 63
Overview 63
ML Pipelines 65
Collecting Data 67
Importance of Data 67
Example Application: Suggesting Runs 68
Responsible Data: Security, Privacy & Fairness 71
Labeling Data 75
Case Study: Degraded Model Performance 75
Data and Concept Change in Production ML 78
Process Feedback and Human Labeling 80
Validating Data 85
Detecting Data Issues 85
TensorFlow Data Validation 89
Week 1 Optional References 93
Week 2: Feature Engineering, Transformation and Selection 95
Feature Engineering 95
Introduction to Preprocessing 95
Preprocessing Operations 97
Feature Engineering Techniques 99
Feature Crosses 104
Feature Transformation at Scale 106
Preprocessing Data at Scale 106
Hello World with tf.Transform 117
Feature Selection 121
Feature Spaces 121
Feature Selection 123
Filter Methods 126
Wrapper Methods 130
Embedded Methods 134
Week 2 Optional References 138
Course #3 Machine Learning Modeling Pipelines in Production 139
Course #4 Deploying Machine Learning Models in Production 140

Course #1 Introduction to Machine Learning in Production

Week 1: Overview of the ML Lifecycle and Deployment

This week covers a quick introduction to machine learning production systems focusing on their requirements and challenges. Next, the week focuses on deploying production systems and what is needed to do so robustly while facing constantly changing data.

Learning Objectives

- **Identify the key components of the ML Lifecycle.**
- **Define "concept drift" as it relates to ML projects.**
- **Differentiate between shadow, canary, and blue-green deployment scenarios in the context of varying degrees of automation.**
- **Compare and contrast the ML modeling iterative cycle with the cycle for deployment of ML products.**
- **List the typical metrics you might track to monitor concept drift.**

Week 2: Select and Train a Model

This week is about model strategies and key challenges in model development. It covers error analysis and strategies to work with different data types. It also addresses how to cope with class imbalance and highly skewed data sets.

Learning Objectives

- **Identify the key challenges in model development.**
- **Describe how performance on a small set of disproportionately important examples may be more crucial than performance on the majority of examples.**
- **Explain how rare classes in your training data can affect performance.**
- **Define three ways of establishing a baseline for your performance.**
- **Define structured vs. unstructured data.**
- **Identify when to consider deployment constraints when choosing a model.**
- **List the steps involved in getting started with ML modeling.**
- **Describe the iterative process for error analysis.**
- **Identify the key factors in deciding what to prioritize when working to improve model accuracy.**
- **Describe methods you might use for data augmentation given audio data vs. image data.**
- **Explain the problems you can have training on a highly skewed dataset.**

- **Identify a use case in which adding more data to your training dataset could actually hurt performance.**
- **Describe the key components of experiment tracking.**

Week 3: Data Definition and Baseline

This week is all about working with different data types and ensuring label consistency for classification problems. This leads to establishing a performance baseline for your model and discussing strategies to improve it given your time and resources constraints.

Learning Objectives

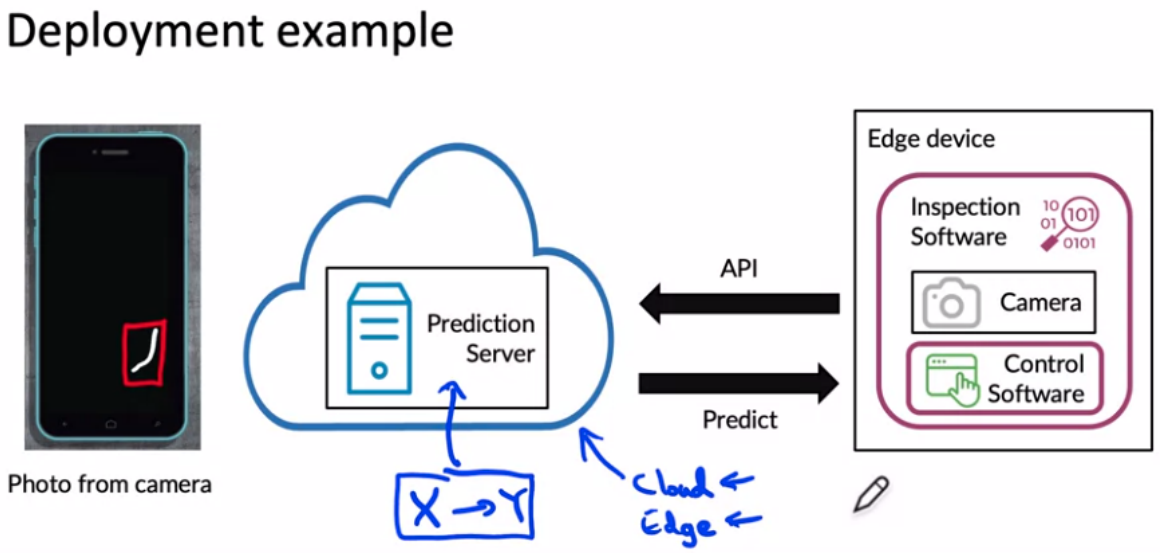
- **List the questions you need to answer in the process of data definition.**
- **Compare and contrast the types of data problems you need to solve for structured vs. unstructured and big vs. small data.**
- **Explain why label consistency is important and how you can improve it**
- **Explain why beating human level performance is not always indicative of success of an ML model.**
- **Make a case for improving human level performance rather than beating it.**
- **Identify how much training data you should gather given time and resource constraints.**
- **Describe the key steps in a data pipeline.**
- **Compare and contrast the proof of concept vs. production phases on an ML project.**
- **Explain the importance of keeping track of data provenance and lineage.**

Week 1: Overview of the ML Lifecycle and Deployment

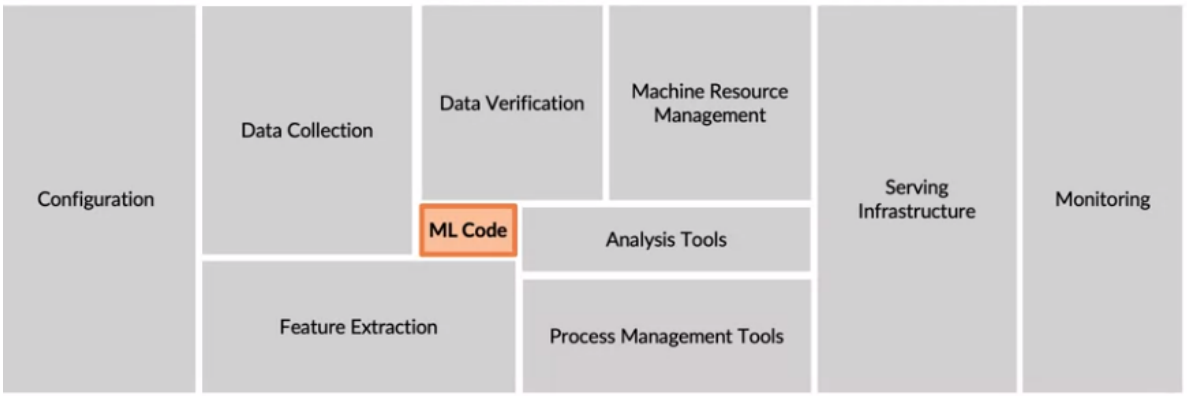
The Machine Learning Project Lifecycle

Just because you've trained a learning algorithm that does well on your test set, which is to be celebrated. It's great when you do well when you hold a test set. Unfortunately reaching that milestone doesn't mean you're done. There can still be quite a lot of work and challenges ahead to get a valuable production deployment running.

For example, let's say your training set has images that look like this. There's a good phone on the left, the one in the middle, it has a big scratch across it and you've trained your learning algorithm to recognize that things like this on the left are okay. Meaning that there are no defects and maybe draw bounding boxes around scratches or other defects that finds and films. When you deploy it in the factory, you may find that the real life production deployment gives you back images like this much darker ones. Because the lighting factory, because the lighting conditions in the factory have changed for some reason compared to the time when the training set was collected. This problem is sometimes called concept drift or data drift.



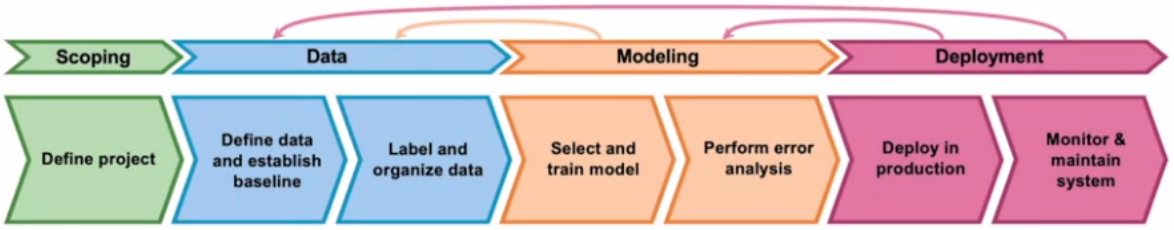
The requirements surrounding ML infrastructure



[D. Sculley et. al. NIPS 2015: Hidden Technical Debt in Machine Learning Systems]

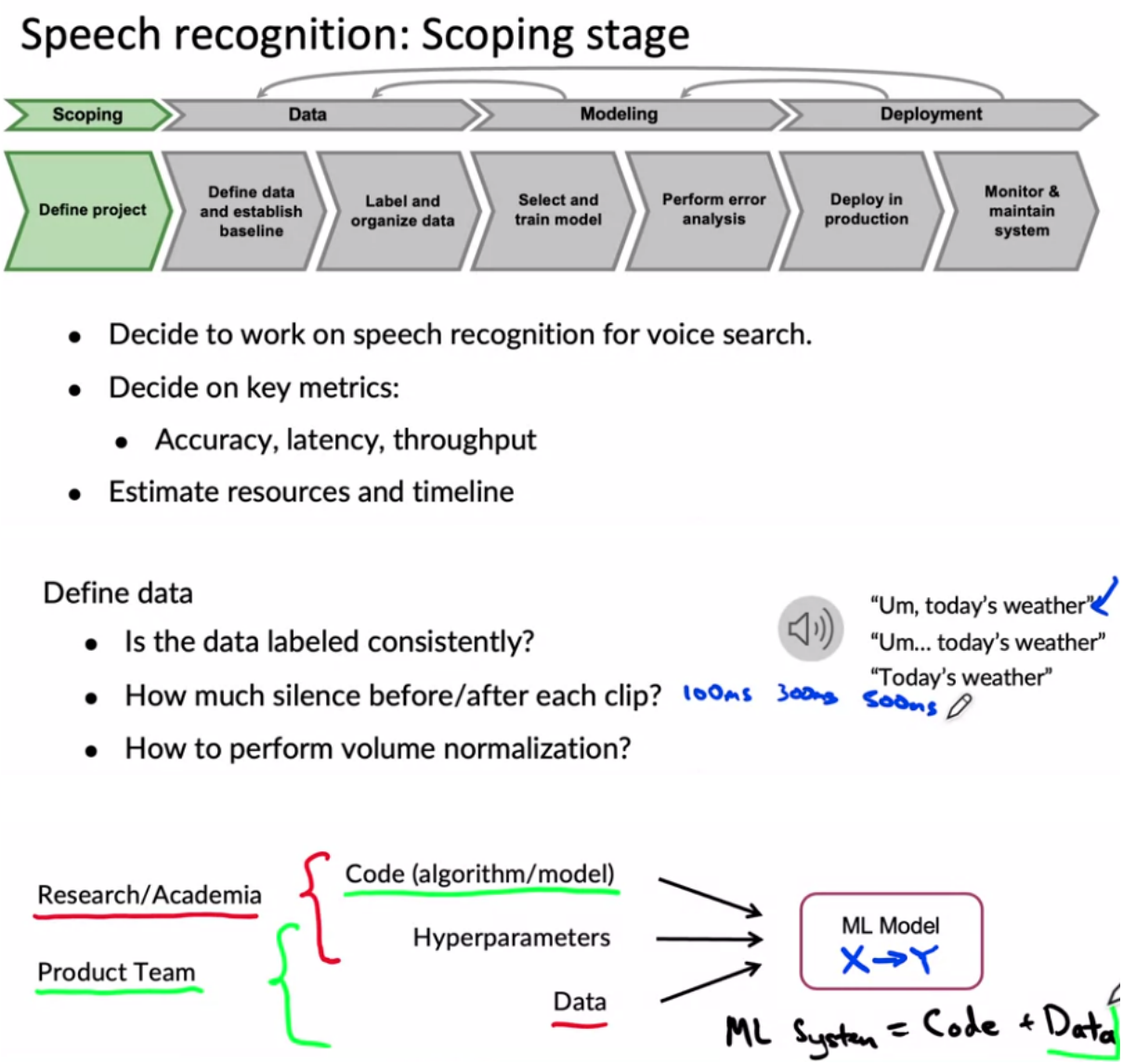
Beyond the machine learning codes there are also many components, especially components for managing the data, such as data collection, data verification, feature extraction.

The ML project lifecycle



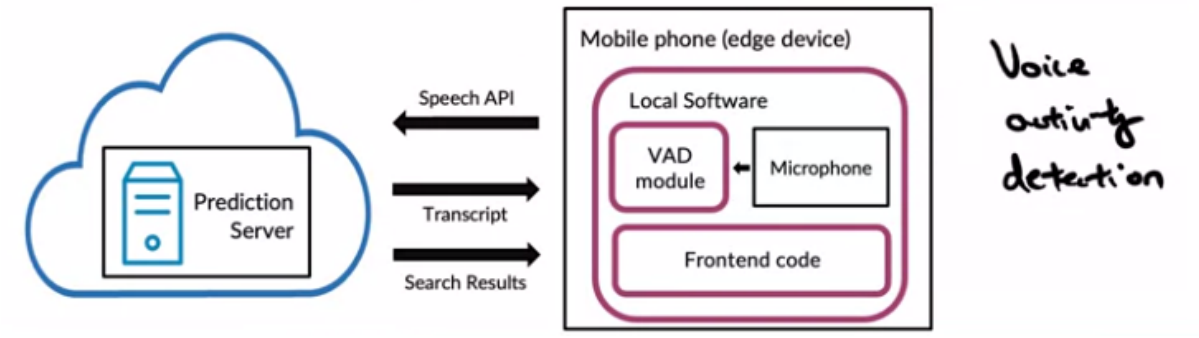
After having chosen the project, you then have to collect data or acquire the data you need for your algorithm.

Now that the system is deployed and is running on live data, and feeding that back into your dataset to then potentially update your data, retrain the model, and so on until you can put an updated model into deployment.

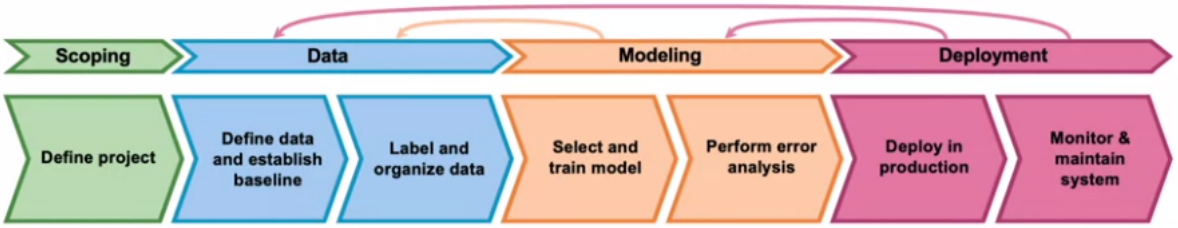


A lot of research work or academic work you tend to hold the data fixed and vary the code and may have varied hyper parameters in order to try to get good performance.

In contrast, for a lot of product teams, if your main goal is to just build and deploy a working, valuable machine learning system, it can be even more effective to hold the code fixed and to instead focus on optimizing the data and maybe the hyper parameters. In order to get a high performing model, A machine learning system includes both codes and data and also hyper parameters that may be a bit easier to optimize than the code or data. And rather than taking a model centric view of trying to optimize the code to your fixed data set for many problems, you can use an open source implementation of something you download from Github and instead just focus on optimizing the data.



Course outline



1. Deployment
2. Modeling
3. Data

Optional: Scoping

MLOps (Machine Learning Operations) is an emerging discipline, and comprises a set of tools and principles to support progress through the ML project lifecycle.

Deployment

Concept drift and Data drift

Speech recognition example

Training set: x → y

- Purchased data, historical user data with transcripts

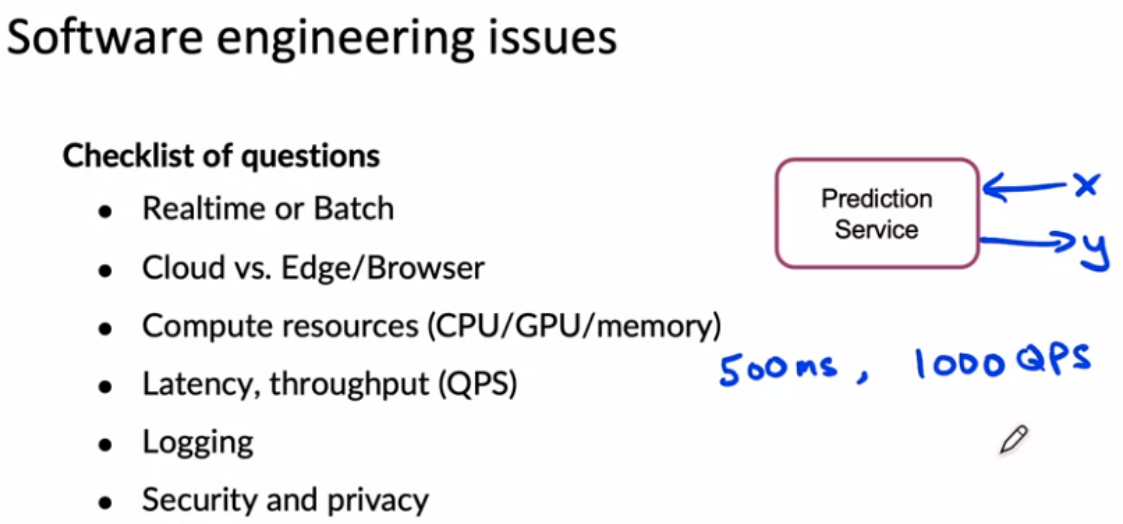
Test set:

- Data from a few months ago

How has the data changed?

Changed: New language/new phone. Gradual change/ Sudden change. Fraud systems failed after covid because people changed their shopping patterns.

Another example of Concept drift, let's say that x is the size of a house, and y is the price of a house, because you're trying to estimate housing prices. If because of inflation or changes in the market, houses may become more expensive over time. The same size house will end up with a higher price. That would be Concept drift. Maybe the size of houses haven't changed, but the price of a given house changes. Whereas data drift would be if, say, people start building larger houses, or start building smaller houses and thus the input distribution of the sizes of houses actually changes over time.



If you save this checklist somewhere, going through this when you're designing your software might help you to make the appropriate software engine choices when implementing your prediction service. To summarize, deploying a system requires two broad sets of tasks: there is writing the software to enable you to deploy the system in production. There is what you need to do to monitor the system performance and to continue to maintain it, especially in the face of concepts drift as well as data drift. One of the things you see when you're building machine learning systems is that the practices for the very first deployments will be quite different compared to when you are updating or maintaining a system that has already previously been deployed.

Deployment patterns

Not just turn on the model and hope for the best

Common deployment cases

1. New product/capability
2. Automate/assist with manual task
3. Replace previous ML system

Key ideas:

- Gradual ramp up with monitoring
- Rollback

1. Example: new recognition system. Pattern: Start with low traffic and then catch up.
2. Already done by a person and we want to automate that task. Example: people in a factory checking subtraction.

Rollback: if the algorithm is not working is nice if we can go back to a previous working version.

Visual inspection example

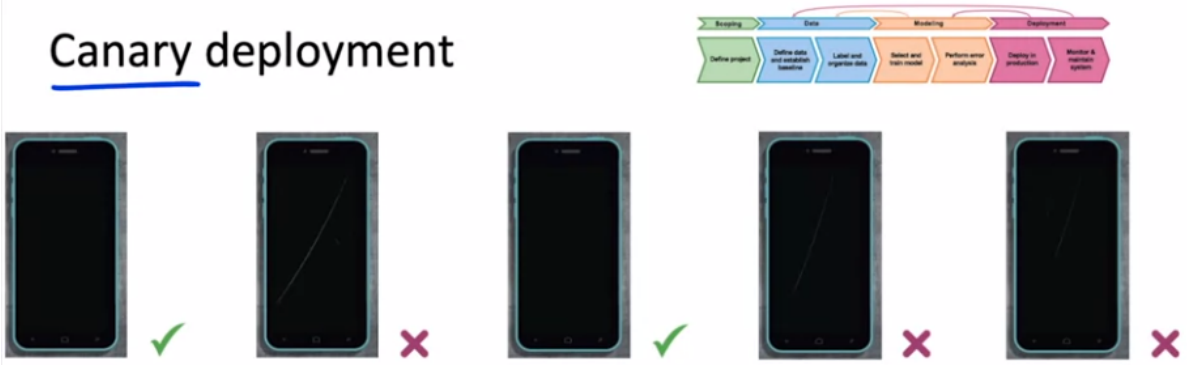
shadow mode

ML system shadows the human and runs in parallel.

ML system's output not used for any decisions during this phase.

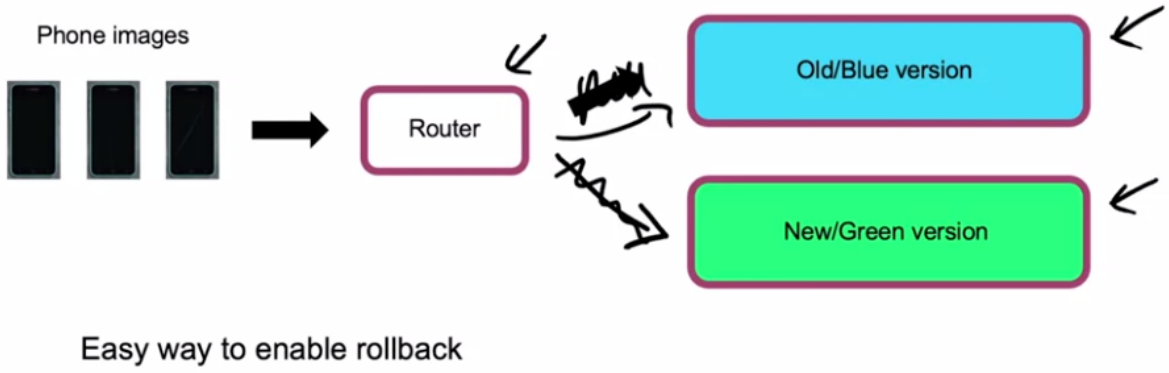
Canary: spot early on problems instead of when we have fully deployed.

Canary deployment



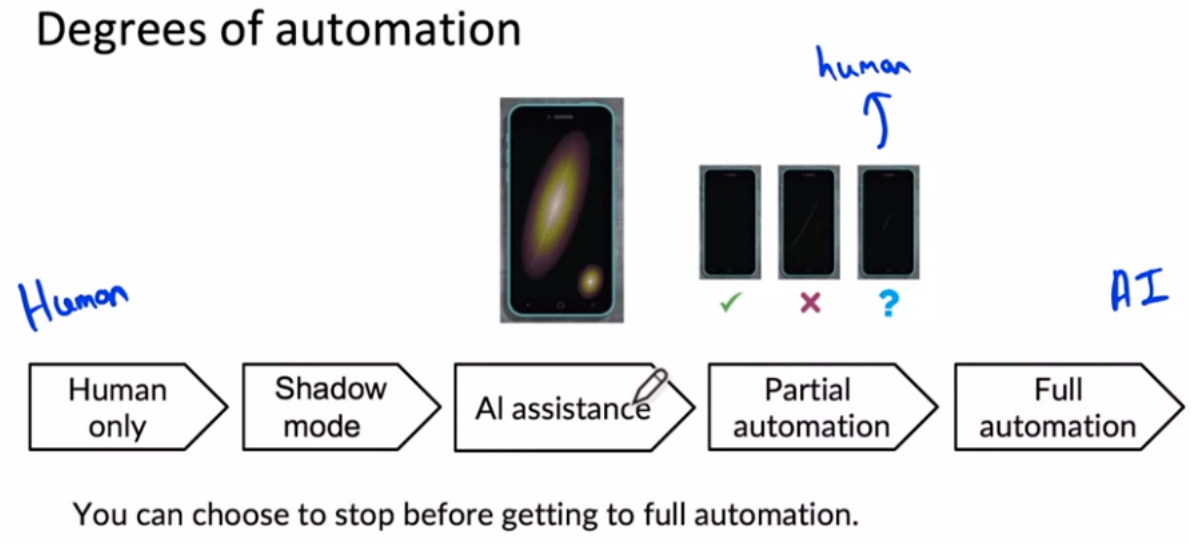
- Roll out to small fraction (say 5%) of traffic initially.
- Monitor system and ramp up traffic gradually.

Blue green deployment

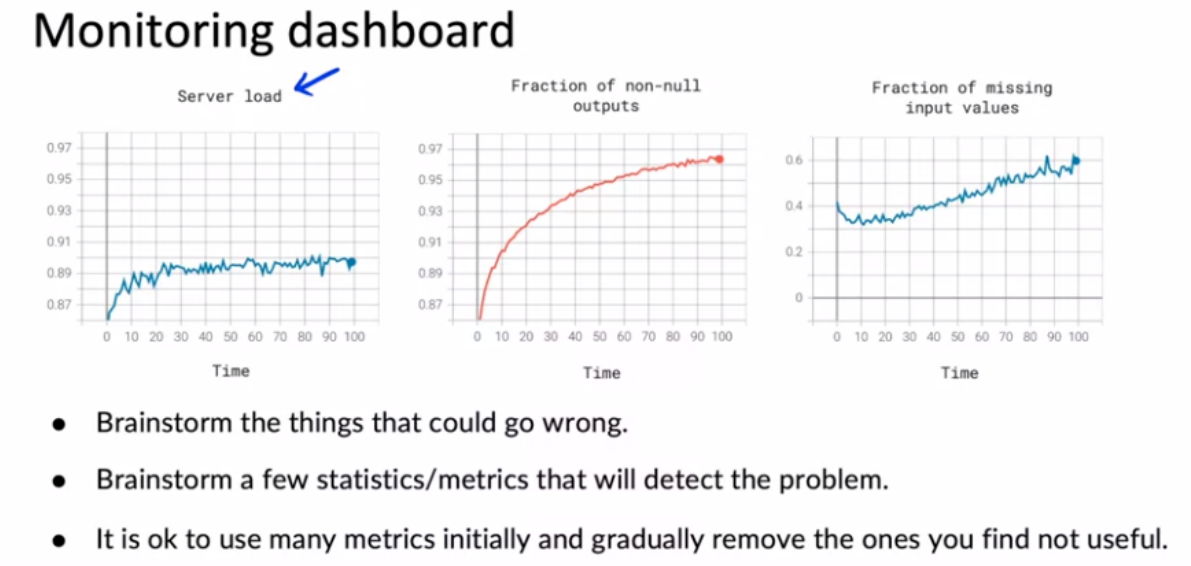


Easy way to enable rollback

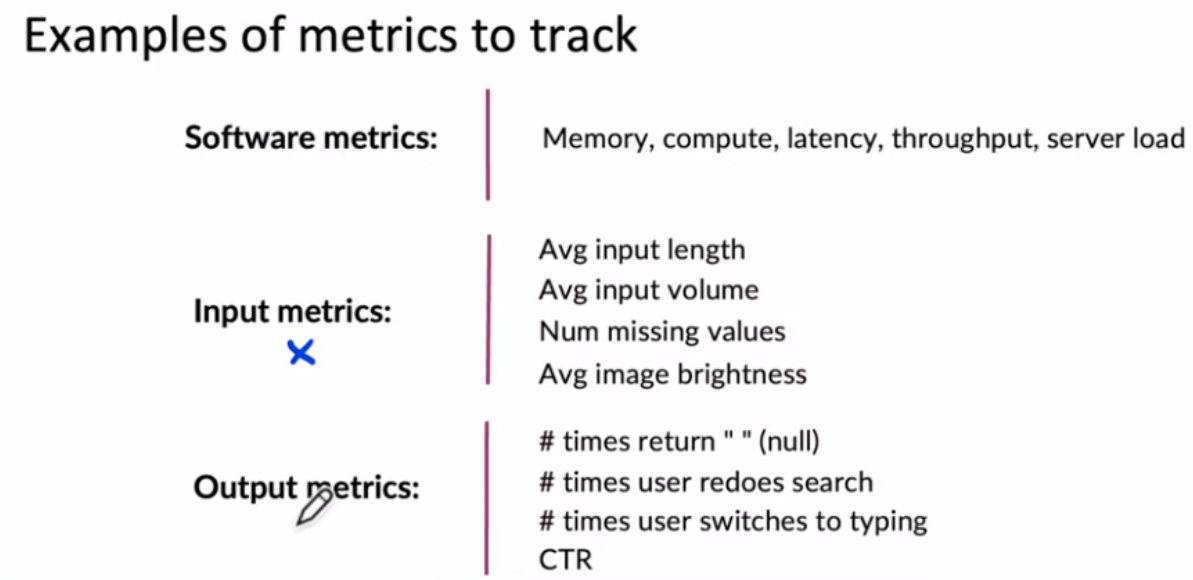
The router changes one deployment or the other



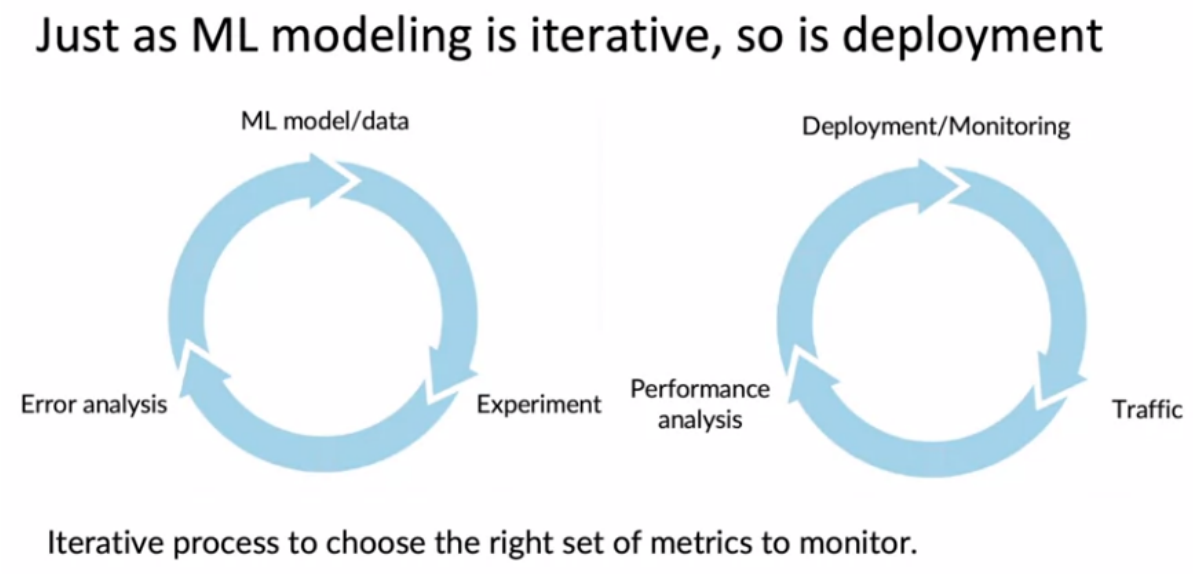
Human in the loop: Steps AI Assistance and Partial Automation



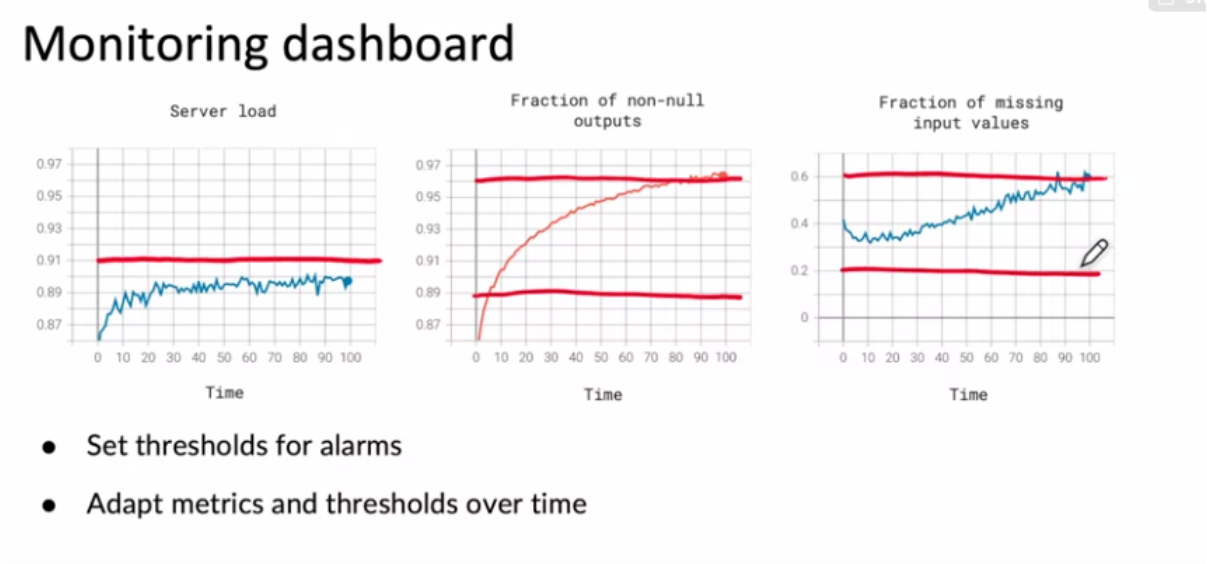
When I'm designing my monitoring dashboards for the first time, I think it's okay to start off with a lot of different metrics and monitor a relatively large set and then gradually remove the ones that you find over time not to be particularly useful.



These output metrics can help you figure out if either your learning algorithm, output y has changed in some way, or if something that comes even after your learning algorithms output, such as the user's switching over to typing has changed in some significant way.

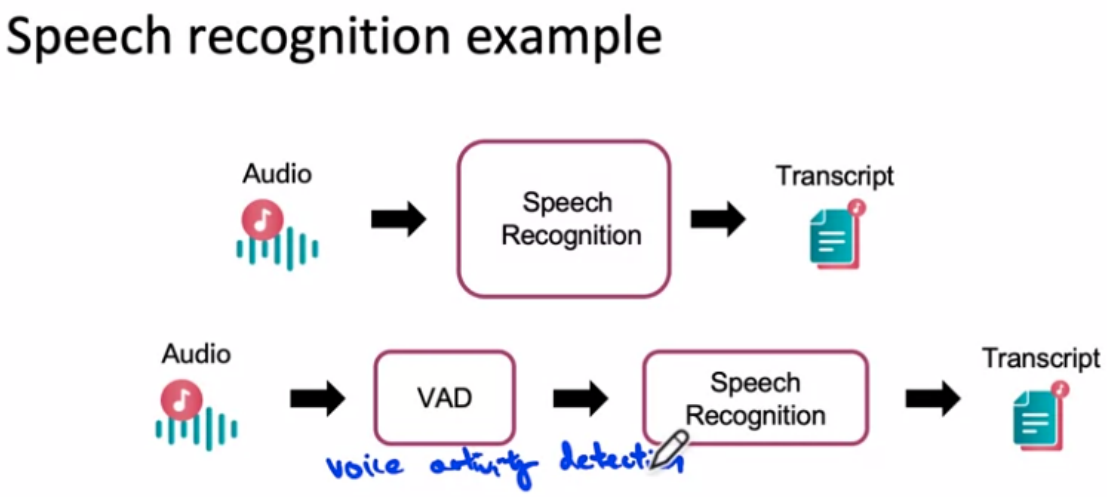


In my experience, it usually takes a few tries to converge to the right set of metrics to monitor.



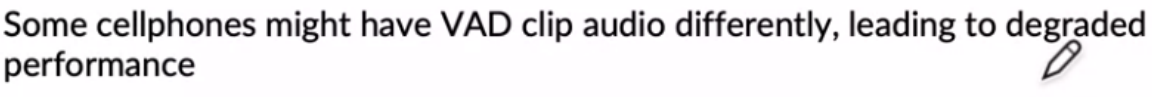
Pipeline Monitoring

Many AI systems are not just a single machine learning model running a prediction service, but instead involves a pipeline of multiple steps. So what are machine learning pipelines and how do you build monitoring systems for that?

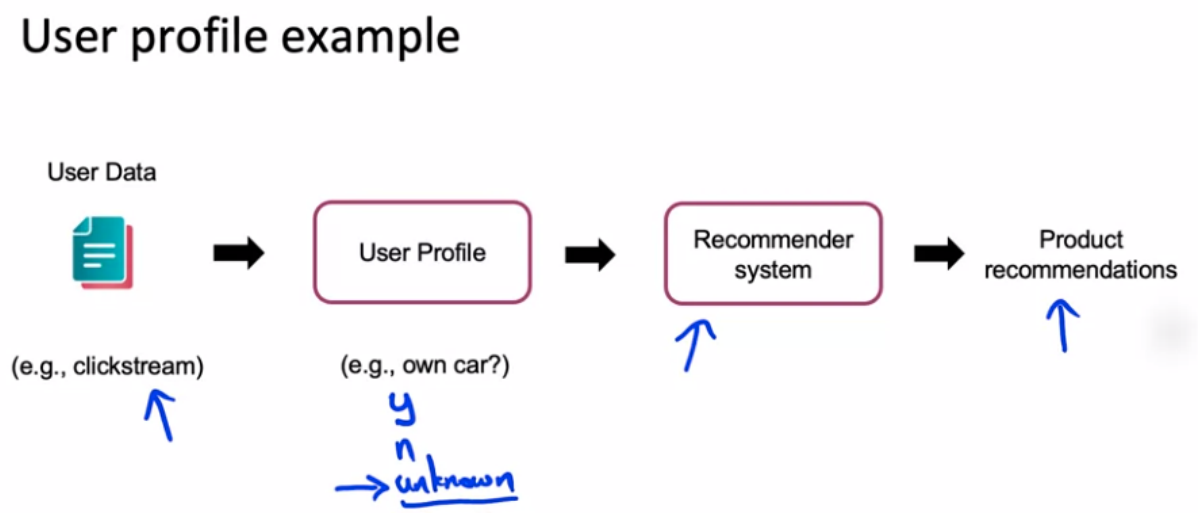


The voice activity detection module looks at the long stream of audio on your cell phone and clips or shortens the audio to just a part where someone is talking and streams only that to the cloud server to perform the speech recognition.

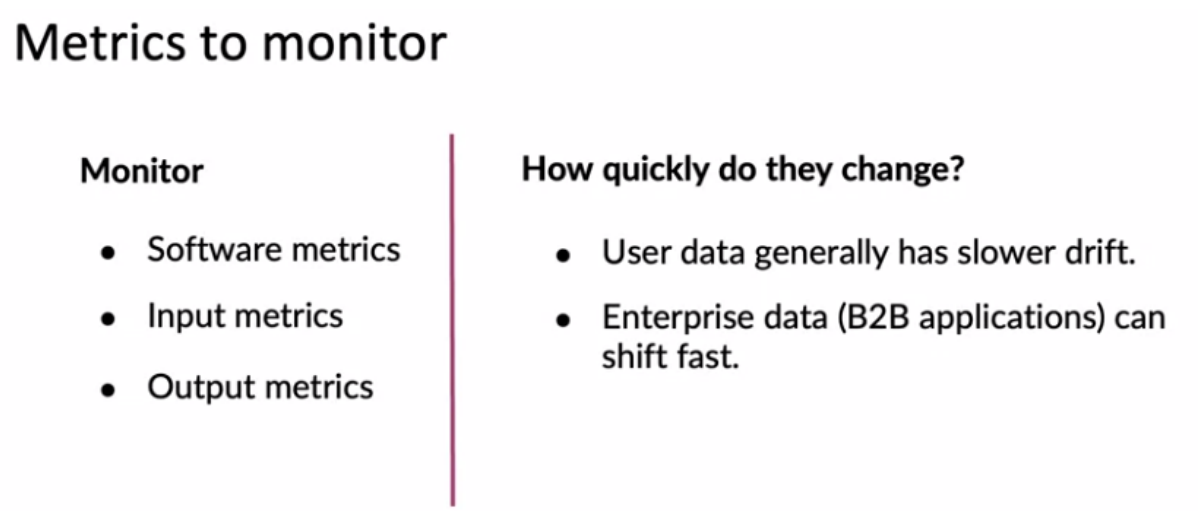
Both VAD and Speech Recognition are ML systems



Maybe it leaves more silence at the start or end or less silence at the start or end and thus if the VAD's output changes, that will cause the speech recognition systems input to change. And that could cause degraded performance of the speech recognition system



When you have a machine learning pipeline, these cascading effects in the pipeline can be complex to keep track of.



Week 1 Optional References

Week 1: Overview of the ML Lifecycle and Deployment

If you wish to dive more deeply into the topics covered this week, feel free to check out these optional references. You won't have to read these to complete this week's practice quizzes.

Concept and Data Drift

Monitoring ML Models

A Chat with Andrew on MLOps: From Model-centric to Data-centric

Papers

Konstantinos, Katsiapis, Karmarkar, A., Altay, A., Zaks, A., Polyzotis, N., … Li, Z. (2020). Towards ML Engineering: A brief history of TensorFlow Extended (TFX). http://arxiv.org/abs/2010.02013

Paleyes, A., Urma, R.-G., & Lawrence, N. D. (2020). Challenges in deploying machine learning: A survey of case studies. http://arxiv.org/abs/2011.09926

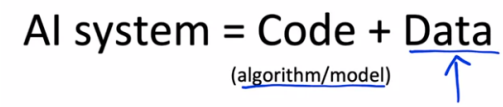
Sculley, D., Holt, G., Golovin, D., Davydov, E., & Phillips, T. (n.d.). Hidden technical debt in machine learning systems. Retrieved April 28, 2021, from Nips.c https://papers.nips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf

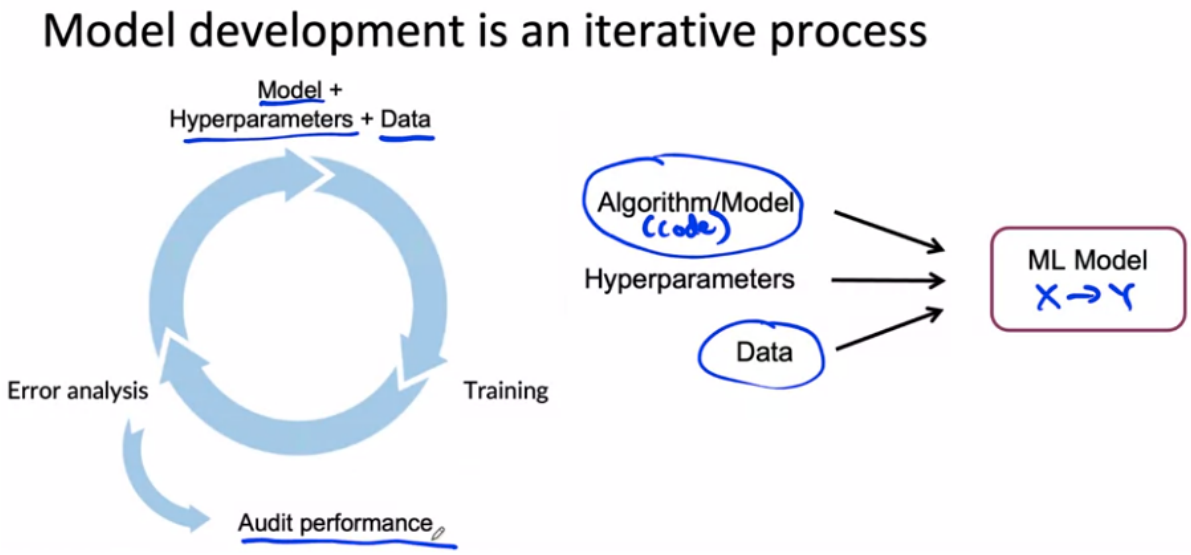
Week 2: Select and Train a Model

This week, our focus will be on the modeling part of the full cycle of a machine learning project, and you learn some suggestions for how to select and train the model, and how to perform error analysis, and use that to drive model improvements.

Selecting and Training a Model

We will fix the model and focus on how we can improve the data in order to obtain better results

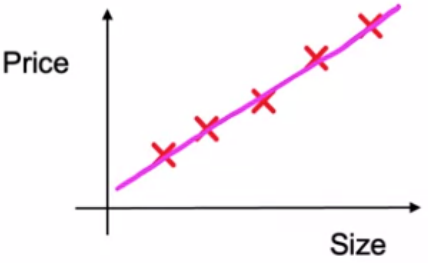




After you've done this enough times and achieve a good model, one last step that's often useful is to carry out a richer error analysis and have your system go through a final audit to make sure that it is working before you push it to a production deployment. So why is model development hard? When building a model, I think there are three key milestones that most projects should aspire to accomplish.

Challenges in model development

1. Doing well on training set (usually measured by average training error).
2. Doing well on dev/test sets.
3. Doing well on business metrics/project goals.



The job of a machine learning engineer would be much simpler if the only thing we ever had to do was do well on the holdout test set. As hard as it is to do well in the holdout test set, unfortunately, sometimes that isn't enough.

Performance on disproportionately important examples

Web Search example

"Apple pie recipe" "Latest movies"
"Wireless data plan" "Diwali festival"
} **Informational and Transactional queries**

"Stanford" "Reddit" "Youtube"
} **Navigational queries**

For informational and transactional queries, a web search engine wants to return the most relevant results, but users are willing to forgive maybe ranking the best result, Number two or Number three. There's a different type of web search query such as Stanford, or Reddit, or YouTube. These are called navigational queries, where the user has a very clear intent, very clear desire to go to Stanford.edu, or Reddit.com, or YouTube.com. When a user has a very clear navigational intent, they will tend to be very unforgiving if a web search engine does anything other than return Stanford.edu as the Number one ranked results and the search engine that doesn't give the right results will quickly lose the trust of its users.

Now one thing you could do is try to give these examples a higher weight. That could work for some applications, but in my experience, just changing the weights of different examples doesn't always solve the entire problem.

Why low average error isn't good enough

Unfortunate conversation in many companies

MLE: "I did well on the test set!"

Product Owner: "But this doesn't work for my application"

MLE: "But... I did well on the test set!"

Performance on key slices of the dataset

Example: ML for loan approval

Make sure not to discriminate by ethnicity, gender, location, language or other protected attributes.

Example: Product recommendations from retailers

Be careful to treat fairly all major user, retailer, and product categories.

Rare classes

Skewed data distribution

99% negative 1% positive

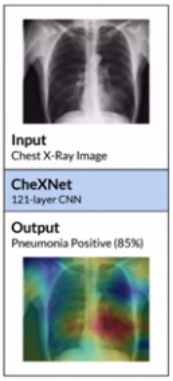
print("0")

Accuracy in rare classes

Condition	Performance
Effusion	0.901
Edema	0.924
Mass	0.909
Hernia	0.851

10,000 → Effusion

~100 → Hernia

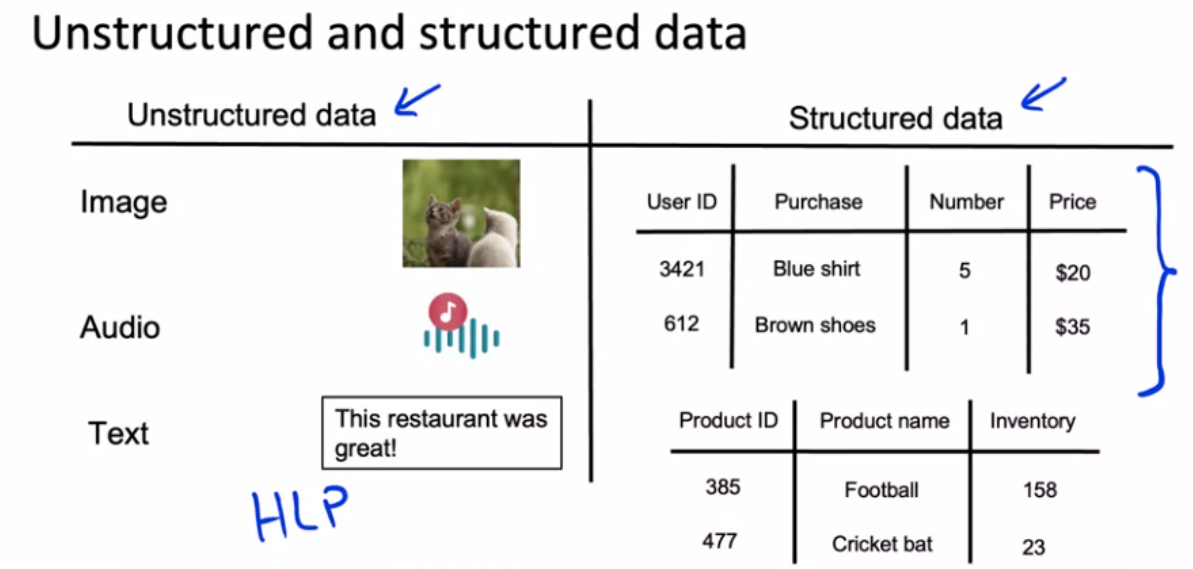


Establishing a baseline level of performance

Speech recognition example:

Type	Accuracy	Human level performance	
Clear Speech	94%	95%	1%
Car Noise	89%	93%	4%
People Noise	87%	89%	2%
Low Bandwidth	70%	70%	~0%

With this analysis, we realized that maybe the low bandwidth audio was so garbled. Even people, humans can't recognize what was said and it may not be that fruitful to work on that. Instead, it may be more fruitful to focus our attention on improving speech recognition with car noise in the background. In this example, using human level performance, which is sometimes abbreviated to HLP, Human Level Performance, gives you a point of comparison or a baseline that helps you decide where to focus your efforts on car noise data rather than on low bandwidth data.



Human level performance (HLP) is generally more effective for establishing a baseline on unstructured data problems (such as images and audio) than structured data problems

Ways to establish a baseline

- Human level performance (HLP)
- Literature search for state-of-the-art/open source
- Quick-and-dirty implementation
- Performance of older system

Baseline helps to indicates what might be possible. In some cases (such as HLP) is also gives a sense of what is irreducible error/Bayes error.

We've talked about how machine learning is an iterative process where you start with a model, data, hyperparameters, training model, carry out error analysis, and then use that to drive further improvements. After you've done this a few times, gone around the loop enough times, when you have a good enough model, you might then carry out a final performance audit before taking it to production. In order to get started on this first step of coming up with the model, here are some suggestions.

Getting started on modeling

- Literature search to see what's possible (courses, blogs, open-source projects).
- Find open-source implementations if available.
- A reasonable algorithm with good data will often outperform a great algorithm with no so good data.

Don't obsess about taking the algorithm that was just published in some conference last week, that is the most cutting edge algorithm, instead find something reasonable, find a good open source implementation and use that to get going quickly. Because being able to get started on this first step of this loop, can make you more efficient in iterating through more times, and that will help you get to good performance more quickly.

Deployment constraints when picking a model

Should you take into account deployment contraints when picking a model?

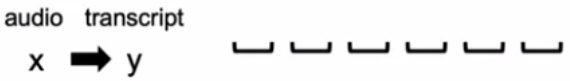
Yes, if baseline is already established and goal is to build and deploy.

No (or not necessarily), if purpose is to establish a baseline and determine what is possible and might be worth pursuing.

Finally, when trying out a learning algorithm for the first time, before running it on all your data, I would urge you to run a few quick sanity checks for your code and your algorithm. For example, I will usually try to overfit a very small training dataset before spending hours or sometimes even overnight or days training the algorithm on a large dataset. Maybe even try to make sure you can fit one training example, especially, if the output is a complex output. For example, I was once working on a speech recognition system where the goal was to input audio and have a learning algorithm output a transcript. When I trained my algorithm on just one example, one audio clip, when I trained my speech recognition system on just one audio clip on the training set, which is just one audio clip, my system outputs this, it outputs space, space, space, space, space, space. Clearly it wasn't working and because my speech system couldn't even accurately transcribe one training example, there wasn't much point to spending hours and hours training it on a giant training set.

Sanity-check for code and algorithm

- Try to overfit a small training dataset before training on a large one.
- Example #1: Speech recognition



- Example #2: Image segmentation

- Example #3: Image classification

10,000
10 , 100

Now, after you've trained a machine learning model, after you've trained your first model, one of the most important things is, how do you carry out error analysis to help you decide how to improve the performance of your algorithm? Let's go on to the next video to dive into error analysis and performance auditing.

Error analysis and performance auditing

Speech recognition example

Example	Label	Prediction	Car noise	People noise	Low bandwidth
1	"Stir fried lettuce recipe"	"Stir fry lettuce recipe"	1		
2	"Sweetened coffee"	"Swedish coffee"		1	1
3	"Sail away song"	"Sell away some"		1	
4	"Let's catch up"	"Let's ketchup"	1	1	1

Until now, error analysis has typically been done via a manual process, say, in the Jupyter notebook or tracking errors in spreadsheets. I still sometimes do it that way and if that's how you're doing it too, that's fine. But there are also emerging MLOps tools that make this process easier for developers. For example, when my team landing AI works on computer vision applications, the whole team now uses Landing Lens, which makes this much easier than the spreadsheet.

You've heard me say that training a model is an iterative process, deploying a model is an iterative process. Maybe it should come as no surprise that error analysis is also an iterative process.

Iterative process of error analysis

Visual inspection:

- Specific class labels (scratch, dent, etc.)
- Image properties (blurry, dark background, light background, reflection, ...)
- Other meta-data: phone model, factory

Product recommendations:

- User demographics
- Product features/category

Useful metrics for each tag

- What fraction of errors has that tag? 12%
- Of all data with that tag, what fraction is misclassified? 18%
- What fraction of all the data has that tag?
- How much room for improvement is there on data with that tag?

So by brainstorming different tags, you can segment your data into different categories and then use questions like these to try to decide what to prioritize working on.

Prioritizing what to work on

Type	Accuracy	Human level performance	Gap to HLP	% of data
Clean Speech	94%	95%	1%	60% → 0.6%
Car Noise	89%	93%	4%	4% → 0.16%
People Noise	87%	89%	2%	30% → 0.6%
Low Bandwidth	70%	70%	0%	6% → ~0%

And so whereas previously we had said there's a lot of room for improvement in car noise, in this slightly richer analysis, we see that because people noise accounts for such a large fraction of the data, it may be more worthwhile to work on either people noise or maybe on clean speech because there's actually larger potential for improvements in both of those than for speech with car noise. So to summarize, when prioritizing what to work on, you might decide on the most

important categories to work on based on, how much room for improvement there is, such as compared to human level performance or according to some baseline comparison.

Prioritizing what to work on

Decide on most important categories to work on based on:

- How much room for improvement there is.
- How frequently that category appears.
- How easy is to improve accuracy in that category.
- How important it is to improve in that category.

Adding/improving data for specific categories

For categories you want to prioritize:

- Collect more data
- Use data augmentation to get more data
- Improve label accuracy/data quality

Data sets where the ratio of positive to negative examples is very far from 50-50 are called skewed data sets. Let's look at some special techniques for handling them.

Examples of skewed datasets

Manufacturing example

99.7% no defect y=0

0.3% defect y=1

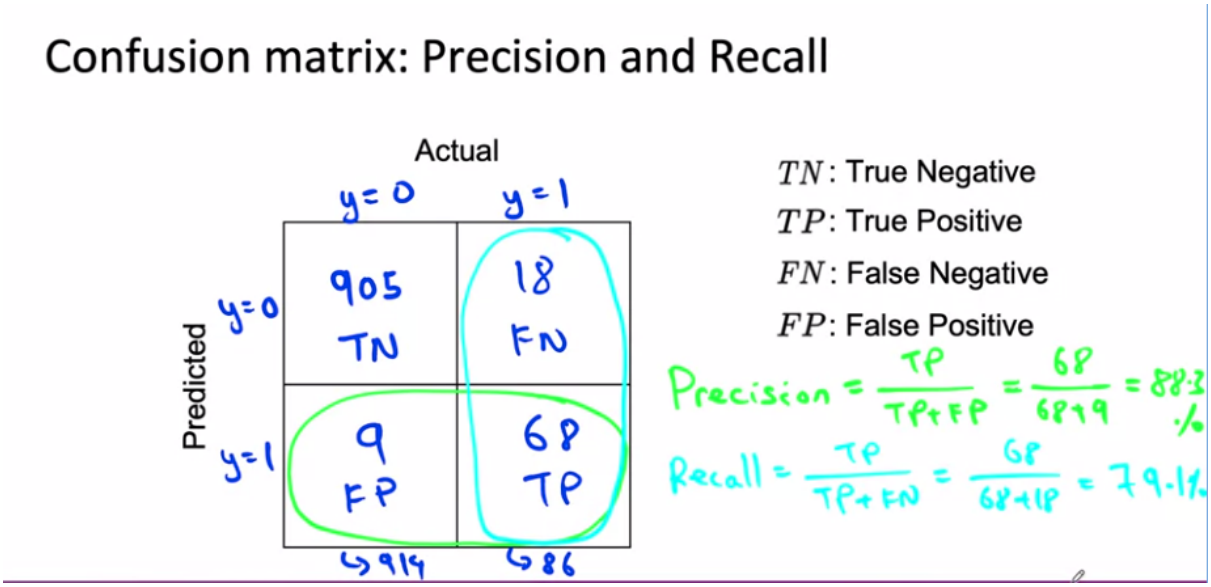
print("0")

99.7%

Medical Diagnosis example: 99% of patients don't have a disease

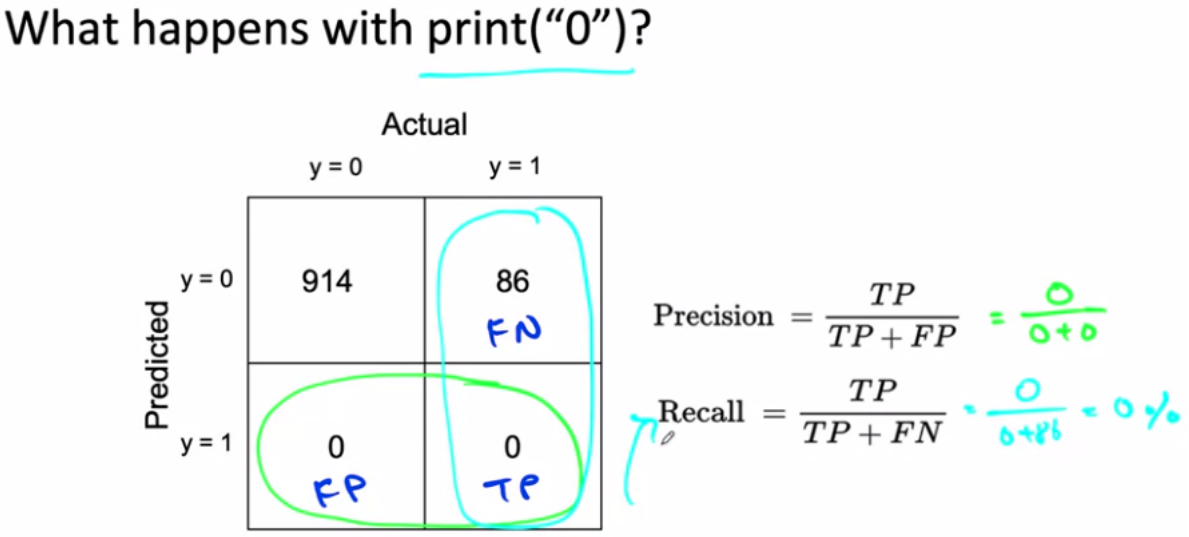
Speech Recognition example: In wake word detection, 96.7% of the time wake word doesn't occur

When you have a very skewed data set like this, raw accuracy is not that useful a metric to look at because Prince Zero can get very high accuracy. Instead, it's more useful to build something called the confusion matrix. A confusion matrix is a matrix where one axis is labeled with the actual label, is the ground truth label, y equals 0 or y equals 1 and whose other axis is labeled with the prediction.



This is indeed a pretty skewed data set where out of 1000 examples there were 940 negative examples and just 86 positive examples, 8.6 percent positive, 91.4 percent negative.

Let's see what happens if your learning algorithm outputs zero all the time. It turns out it won't do very well on recall.



Combining precision and recall – F_1 score

	Precision (P)	Recall (R)	F_1
Model 1	88.3	79.1	83.4% ←
Model 2	97.0	7.3	13.6%

$$F_1 = \frac{2}{\frac{1}{P} + \frac{1}{R}}$$

Being F1 the harmonic mean of Precision and recall, the score punished the model if one of those metrics gave a low score.

So far, we've talked about the binary classification problem with skewed data sets. It turns out to also frequently be useful for multi-class classification problems.

Multi-class metrics

Classes: Scratch, Dent, Pit mark, Discoloration

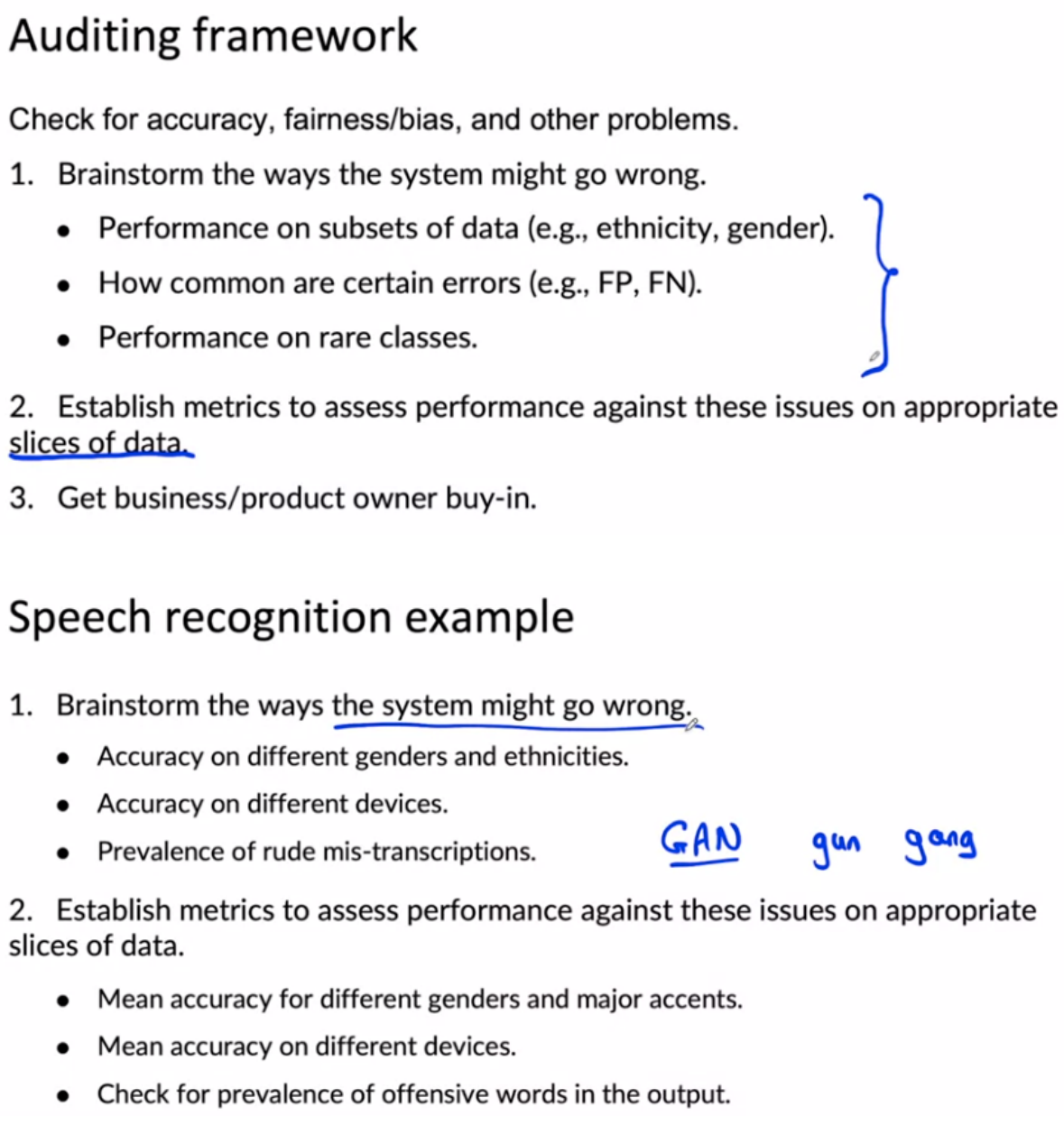
Defect Type	Precision	Recall	F_1
Scratch	82.1%	99.2%	89.8%
Dent	92.1%	99.5%	95.7%
Pit mark	85.3%	98.7%	91.5%
Discoloration	72.1%	97%	82.7%

By combining precision and recall using F_1 as follows, this gives you a single number evaluation metric for how well your algorithm is doing on the four different types of defects and can also help you benchmark to human-level performance and also prioritize what to work on next. Instead of accuracy on scratches, dents, pit marks, and discolorations, using F_1 score can help you to prioritize the most fruitful type of defect to try to work on. The reason we use F_1 is

because, maybe all four defects are very rare and so accuracy would be very high even if the algorithm was missing a lot of these defects. I hope that these tools will help you both evaluate your algorithm as well as prioritize what to work on, both in problems with skewed data sets and for problems with multiple rare classes.

Performance auditing

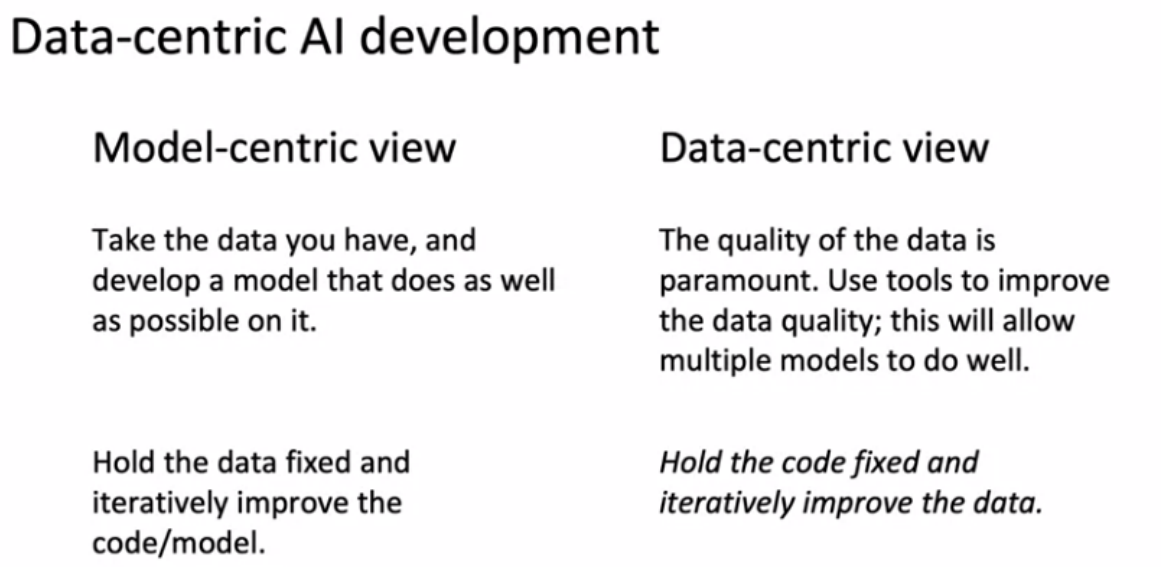
Even when your learning algorithm is doing well on accuracy or F1-score or some appropriate metric is often worth one last performance audit before you push it to production. And this can sometimes save you from significant post deployment problems.



Data iteration

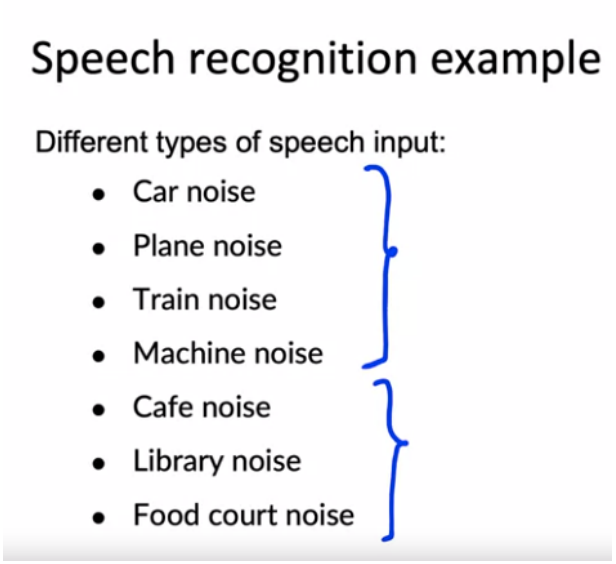
With a model centric view of AI developments, you would take the data you have and then try to work really hard to develop a model that does as well as possible on the data because a lot of academic research on AI was driven by researchers, downloading a benchmark data set and trying to do well on that benchmark. Most academic research on AI is model centric, because the benchmark data set is a fixed quantity. In this view, model centric development, you would hold the data fixed and iterate the improvement. In this model centric view, you would hold the data fixed and iteratively improve the code or the model.

Which is to shift data from a model centric view to a data centric view. In this view, we think of the quality of the data as paramount, and you can use tools such as era analysis or data augmentation to systematically improve the data quality. For many applications, I find that if your data is good enough, there are multiple models that will do just fine. In this view, you can instead hold the code fixed and iteratively improve the data.

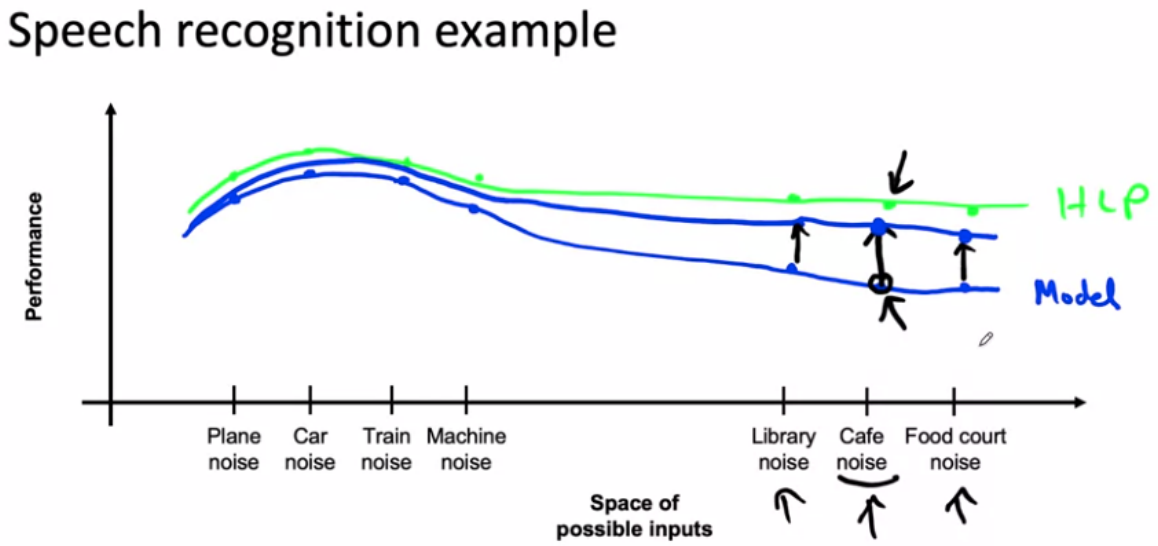


If you've been used to model centric thinking for most of your experience with machine learning, I would urge you to consider taking a data centric view as well, where when you're trying to improve your learning outcomes performance, try asking how you can make your data set even better?

A useful picture of data augmentation



There are two types of noise: ones that come with mechanical noise and ones with human noise.



So this gap represents an opportunity for improvement. Now, what happens if you use data augmentation or maybe not data augmentation but go out to a bunch of actual cafes, to collect a lot more data with cafe noise in the background. What you'll do is, you'll take this point imagine grabbing a hold of these blue rubber bands or this rubber sheet, and pulling it upward like so. That's what you're doing if you collect or somehow get more data with cafe noise and add that to your training set, you're pulling up the performance of the algorithm on inputs with cafe noise. And what that will tend to do, is pull up this rubber sheet in the adjacent region as well. So if performance on cafe noise goes up, probably performance on the nearby points will go up to and performance on far away. Points may or may not go up as much. It turns out that for unstructured data problems, pulling up one piece of this rubber sheet is unlikely to cause a different piece of the rubber sheet to dip down really far below. Instead, pulling up one point causes nearby points to be pulled up quite a lot and far away points may be pulled up a little bit, or if you're lucky, maybe more than a little bit.

Data augmentation

Goal:

Create realistic examples that (i) the algorithm does poorly on, but (ii) humans (or other baseline) do well on

Checklist:

- Does it sound realistic?
- Is the x → y mapping clear? (e.g., can humans recognize speech?)
- Is the algorithm currently doing poorly on it?

Let's say that you have a very small set of images of smartphones with scratches. Here's how you may be able to use data augmentation. You can take the image and flip it horizontally. This results in a pretty realistic image. The phone buttons are now on the other side, but this could be a useful example to add to your training set. Or you could implement contrast changes or actually brighten up the image here so the scratch is a little bit more visible. Or you could try darkening the image, but in this example, the image is now so dark that even I as a person can't really tell if there's a scratch there or not. Whereas these two examples on top would pass the checklist we had earlier, that the human can still detect the scratch well, this example is too dark, it would fail that checklist. I would try to choose the data augmentation scheme that generates more examples that look like the ones on top and fill the ones that look like the ones here at the bottom.

If you're using data augmentation, you're adding to specific parts of the training set such as adding lots of data with cafe noise. So now your training set may come from a very different distribution than the death set and the test set. Is this going to hurt your learning album's performance?

Can adding data hurt performance?

For unstructured data problems, if:

- The model is large (low bias).
- The mapping x → y is clear (e.g., given only the input x, humans can make accurate predictions).

Then, **adding data rarely hurts accuracy.**

Photo OCR counterexample

Adding a lot of new "I"s may skew the dataset and hurt performance

If we want to identify the "I"s we can use data augmentation, but by adding new examples the third option can be considered an "I" when, as being ambiguous, it should be considered a 1.

For many structured data problems. It turns out that creating brand new training examples is difficult, but there's something else you could do which is to take existing training examples and figure out if there are additional useful features you can add to it.

Structured data

Restaurant recommendation example

Vegetarians are frequently recommended restaurants with only meat options.

Possible features to add?

- Is person vegetarian (based on past orders)?
- Does restaurant have vegetarian options (based on menu)?

Additional features like these, can be hand coded or they could in turn be generated by some learning algorithm, such as having a learning average home, try to read the menu and classify meals as vegetarian or not, or having people call this manually could also work depending on your application.

Other food delivery examples

- Only tea/coffee
- Only pizza

What are the added features that can help make a decision?

Product recommendation:

Collaborative filtering → Content based filtering Cold-start

Over the last several years, there's been a trend in product recommendations of a shift from collaborative filtering approaches to what content based filtering approaches. Collaborative filtering approaches is loosely an approach that looks at the user, tries to figure out who is similar to that user and then recommends things to you that people like you also liked. In contrast, a content based filtering approach will tend to look at you as a person and look at the description of the restaurant or look at the menu of the restaurants and look at other information about the restaurant, to see if that restaurant is a good match for you or not. The advantage of content based filtering is that even if there's a new restaurant or a new product that hardly anyone else has liked by actually looking at the description of the restaurant, rather than just looking at who else likes the restaurants, you can more quickly make good recommendations. This is sometimes also called the Cold Start Problem. How do you recommend a brand new product that almost no one else has purchased or likes or dislikes so far? And one of the ways to do that is to make sure that you capture good features for the things that you might want to recommend. Unlike collaborative filtering, which requires a bunch of people to look at the product and decide if they like it or not, before it can decide whether a new user should recommend the same product.

- Error analysis can be harder if there is no good baseline (such as HLP) to compare to.
- Error analysis, user feedback and benchmarking to competitors can all provide inspiration for features to add.

As you're working to iteratively improve your algorithm. One thing that'll help you be a bit more efficient is to make sure that you have robust experiment tracking.

Experiment tracking

What to track?

- Algorithm/code versioning
- Dataset used
- Hyperparameters
- Results

Tracking tools

- Text files
- Spreadsheet
- Experiment tracking system

Desirable features

- Information needed to replicate results
- Experiment results, ideally with summary metrics/analysis
- Perhaps also: Resource monitoring, visualization, model error analysis

Rather than worrying too much about exactly which experiment tracking framework to use though, the number one thing I hope you take away from this video is, do got to have some system, even if it's just a text file or just a spreadsheet for keeping track of your experiments and include as much information as is convenient to include. Because later on, if you try to look back, remember how you had generated a certain model, having that information would be really useful for helping you to replicate your own results.

I'd like to leave you with a thought on shifting from big data to good data.

From Big Data to Good Data

Try to ensure consistently high-quality data in all phases of the ML project lifecycle.

Good data:

- Covers important cases (good coverage of inputs x) ←
- Is defined consistently (definition of labels y is unambiguous)
- Has timely feedback from production data (distribution covers data drift and concept drift)
- Is sized appropriately

Week 2 Optional References

Week 2: Select and Train Model

If you wish to dive more deeply into the topics covered this week, feel free to check out these optional references. You won't have to read these to complete this week's practice quizzes.

Establishing a baseline

Error analysis

Experiment tracking

Papers

Brundage, M., Avin, S., Wang, J., Belfield, H., Krueger, G., Hadfield, G., … Anderljung, M. (n.d.). Toward trustworthy AI development: Mechanisms for supporting verifiable claims∗. Retrieved May 7, 2021http://arxiv.org/abs/2004.07213v2

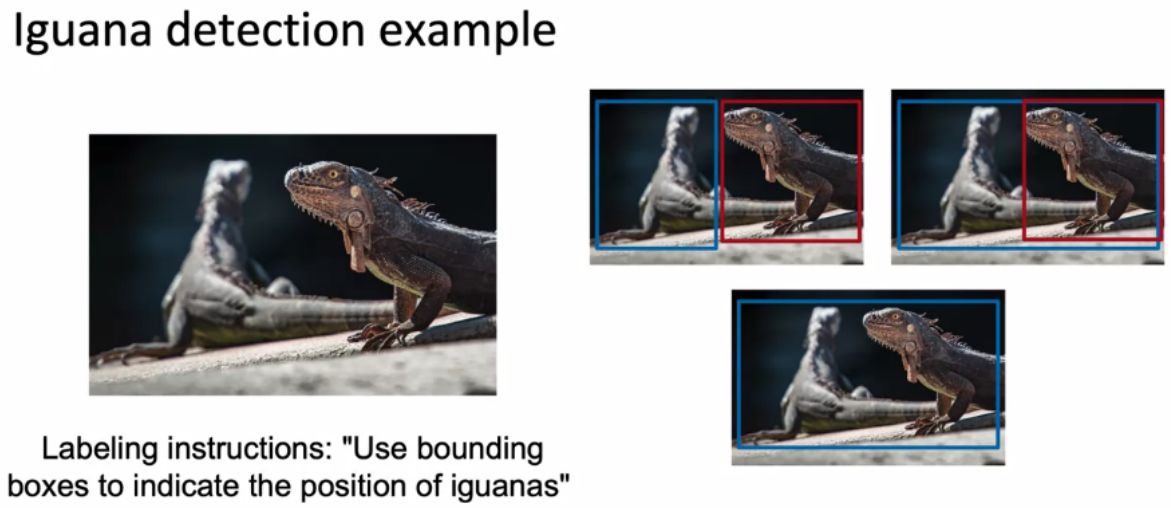
Nakkiran, P., Kaplun, G., Bansal, Y., Yang, T., Barak, B., & Sutskever, I. (2019). Deep double descent: Where bigger models and more data hurt. Retrieved from http://arxiv.org/abs/1912.02292

Week 3: Data Definition and Baseline

This week we dive into data. How do you get data that sets up your training, your modeling for success?

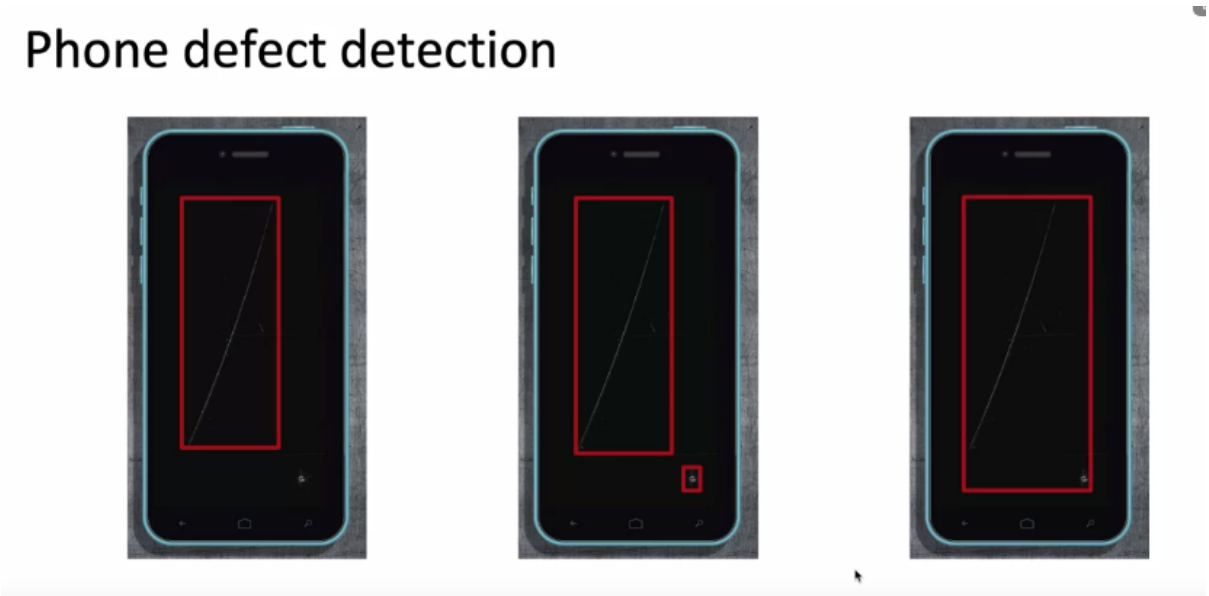
Define Data and Establish Baseline

Why is data definition hard?



Three diligent, hard working laborers can come up with these three very different ways of labeling iguana's, and maybe any of these is actually fine. I would prefer the top two rather than the 3rd one. But any of these labeling conventions could result in you learning algorithm, learning a pretty good iguana detector.

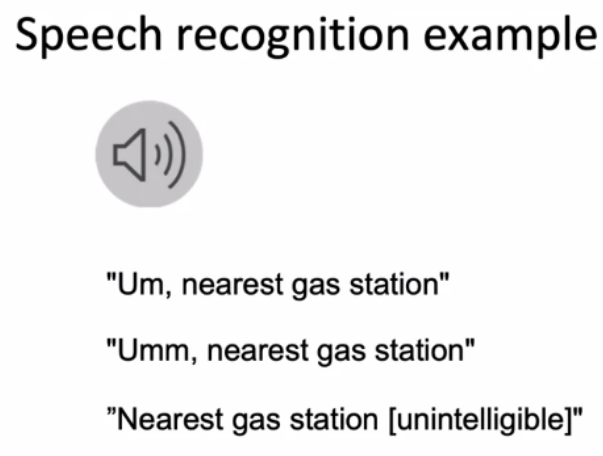
But what is not fine is if 1/3 of your laborers use the 1st and 1/3 the 2nd, and 1/3, the 3rd labeling convention, because then your labels are inconsistent, and this is confusing to the learning algorithm.



I think the 2nd laborer probably did a better job. But then a 3rd laborer may look at this and say, well, here's a bounding box that shows you where the defects are. Between these three labels, probably the one in the middle would work the best.

What we would do this week is dive into best practices for the data stage of the full cycle of a machine learning project. Specifically, we'll talk about how to define what is the data, what should be x and what should be y and establish a baseline and doing that well will set you up to label and organize the data well, which would give you a good data set for when you move into the modeling phase.

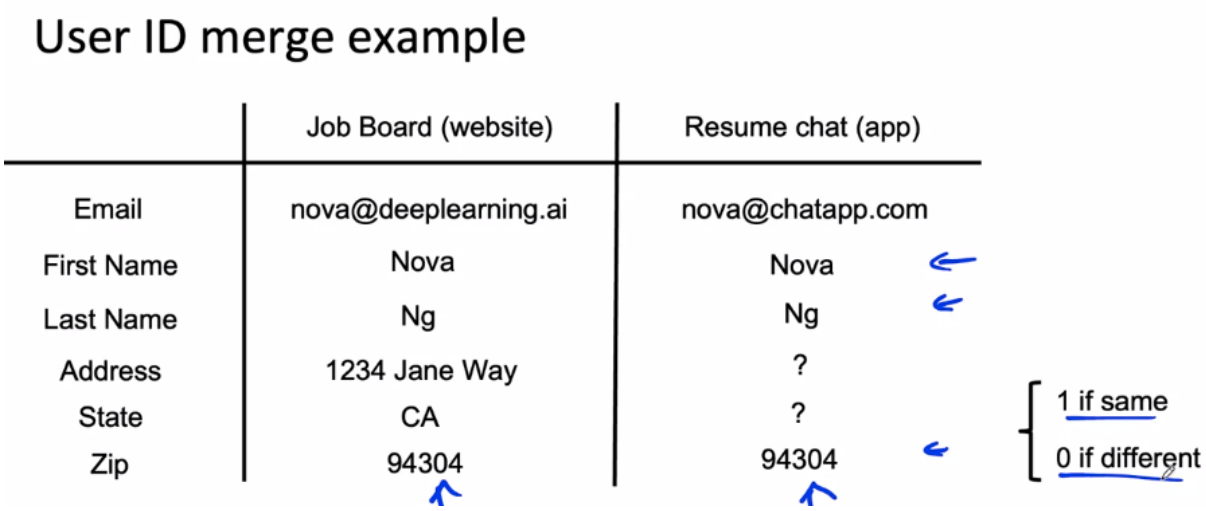
More label ambiguity examples



One way to transcribe this would be "Um, nearest gas station." In some places, people spell "um" with two m's. That would be a different way to spell it. We could have used dot-dot-dot or ellipses instead of the comma as well, which would be another ambiguity. Or given the audio had noise after the last words. Nearest gas station. Did they say something after the nearest gas station? I'm not sure actually. Would you transcribe it like this instead? There are combinatorially many ways to transcribe this. With one M or two M's, comma or ellipses, whether to write unintelligible at the end of this. Being able to standardize on one convention will help your speech recognition algorithm.

Let's also look at an example of structured data.

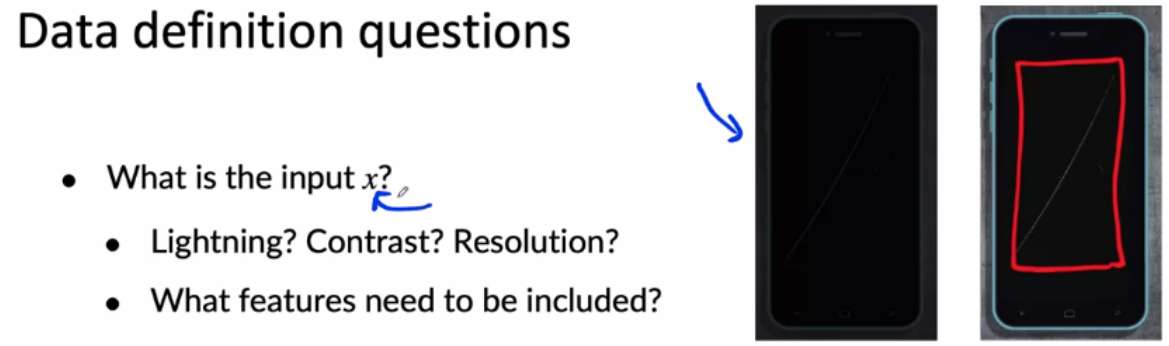
A common application in many large companies is user ID merge. That's when you have multiple data records that you think correspond to the same person and you want to merge these user data records together.



Now, say your company acquires a second company that runs a mobile app that allows people to login, to chat and get advice from each other about their resumes. It seems synergistic for your business. If you run a listing of online jobs, maybe you merge or acquire a second company that runs a mobile app that lets people chat about their resumes and from this mobile app, you have a different database of users. Given this data record and this one, do you think these two are the same person? One approach to the User ID merge problem, is the use of supervised learning algorithm that takes as inputs to user data records and tries to output either one or zero based on whether it thinks these two are actually the same physical human being. If you have a way to get ground truth data records, such as if a handful of users are willing to explicitly link the two accounts, then that could be a good set of labeled examples to train an algorithm. But if you don't have such a ground true set of data, what many companies have done is ask human labors, sometimes a product management team to just manually look at some pairs of records that have been filtered to have maybe similar names or similar ZIP codes, and then to just use human judgment to determine if these two records appear to be the same person. Because whether these two records really are the same person, is genuinely ambiguous.

A few other examples from structured data. If you are trying to use the learning algorithm to look at the user account like this and predict is it a bot or a spam account?

Sometimes that can be ambiguous. Or if you look at an online purchase, is this a 40-length transaction? Has someone stolen accounts and is using stolen accounts to interact with your websites or to make purchases? Sometimes that too is ambiguous. Or if you look at someone's interactions with your website and you want to know, are they looking for a new job at this moment in time based on how someone behaves on a job board website or a resume chat app, you can sometimes guess if they're looking for a job, but it's hard to be sure. That's also a little bit ambiguous. In the face of potentially very important and valuable prediction tasks like these, the ground truth can be ambiguous. If you ask people to take their best guess at the ground truth label for tasks like these, giving labeling instructions that results in more consistent and less noisy and less random labels will improve the performance of your learning algorithm.

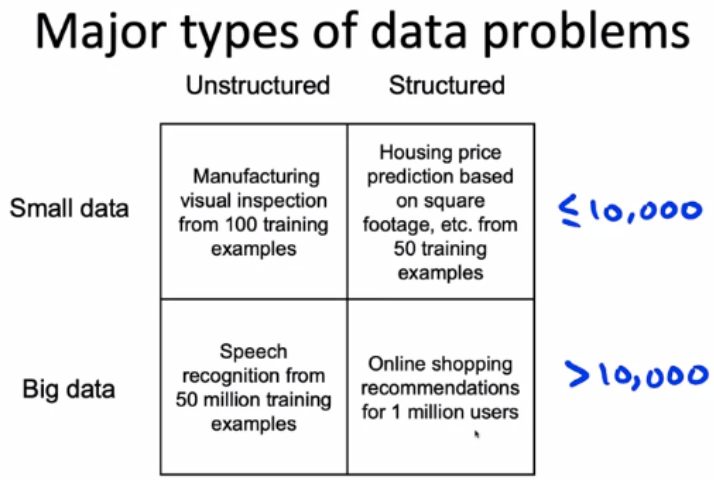


When defining the data for your learning algorithm, here are some important questions. First, what is the input x? For example, if you are trying to detect defects on smartphones, for the pictures you're taking, is the lighting good enough? Is the camera contrast good enough? Is the camera resolution good enough? If you find that you have a bunch of pictures like these, which are so dark, it's hard even for a person to see what's going on. The right thing to do may not be to take this input x and just label it. It may be to go to the factory and politely request improving the lighting because it is only with this better image quality that the labor can then more easily see scratches like this and label them.

In addition to defining the input x, you also have to figure out what should be the target label y. As you've seen from the preceding examples, one key question is, how can we ensure labels give consistent labels?

Major types of data problems

It turns out that the best practices for organizing data for one type can be quite different than the best practices for totally different types.



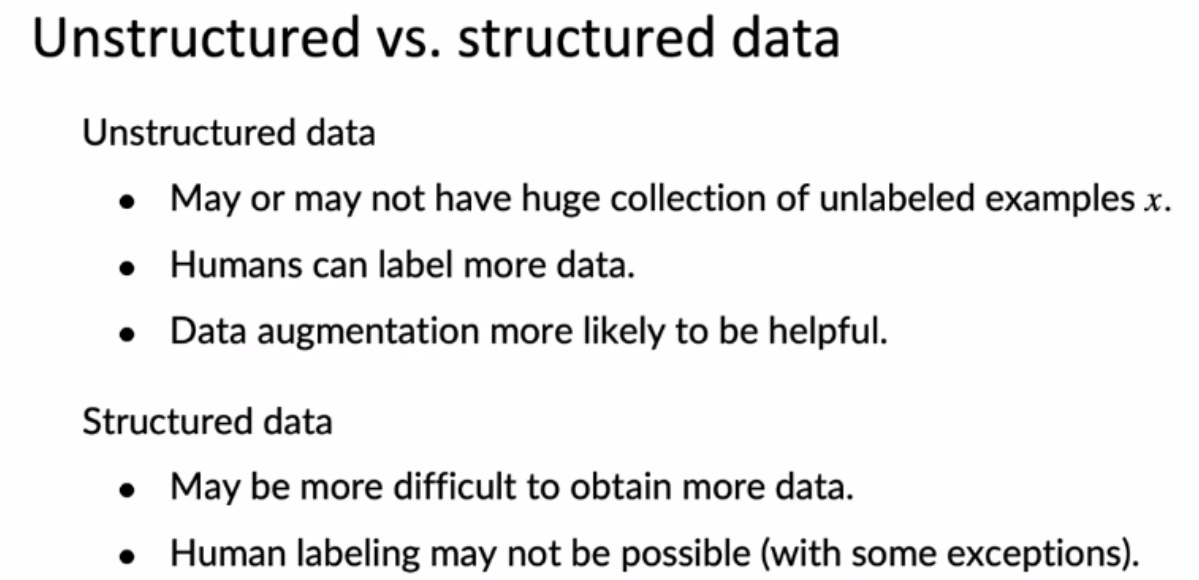
Do you have a relatively small data set? Do you have relatively small datasets or do you have a very large data set? There is no precise definition of what exactly is small and what is large? But I'm going to use it as a slightly arbitrary threshold, whether you have over 10,000 examples or not. And clearly this boundary is a little bit fuzzy and the transitions from small to big data sets is a gradual one.

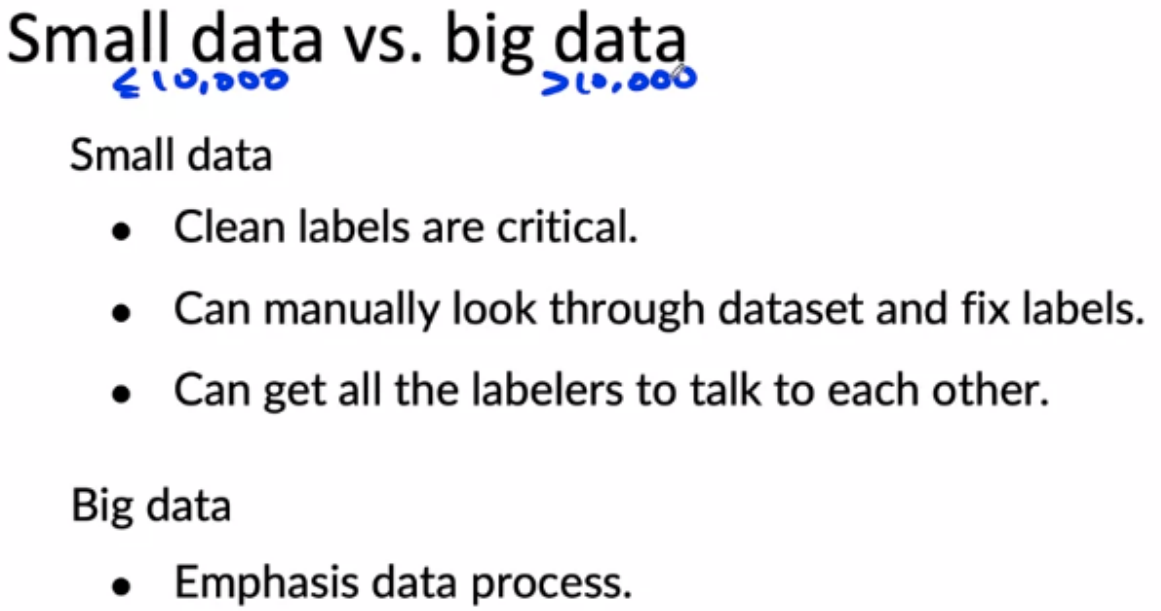
And the reason I chose the number 10,000 is that's roughly

the size beyond which it becomes quite painful to examine every single example yourself.

For a lot of **unstructured data problems**, people, Can help you to label data and data augmentation such as synthesizing new images or synthesizing new audio. And there's some emerging techniques for synthesizing new text as well, but data augmentation can help. So for manufacturing vision inspection, you can use data augmentation to maybe generate more pictures of smart films or for speech recognition. Data augmentation can help you synthesize audio clips with different background noise. In contrast for **structured data problems**, it can be harder to obtain more data and also harder to use data augmentation, if only 50 houses have been so recently in that geography. Well, it's hard to synthesize new houses that don't exist or if you have a million users in your database, again, it's hard to synthesize new users that don't really exist. And it's also harder not impossible, still worth trying, but it may or may not be possible to get humans to label the data. So I find that the best practices for unstructured versus structured data are quite different.

The second axis is the size of your data set. When you have a relatively small data set, having clean labels is critical. If you have 100 training examples, then if just one of the examples is mislabeled, that's 1% of your data set. And because the data set is small enough for you or a small team to go through it efficiently, it may well be aware of your while to go through that 100 examples. And make sure that every one of those examples is labelled in a clean and consistent way, meaning according to a consistent labeling standard. In contrast, if you have a million data points, it can be harder. Maybe impossible for a small machine learning team to manually go through every example. Having clean labels is still very helpful, don't get me wrong. Even when you have a lot of data, clean labels are better than non clean ones.



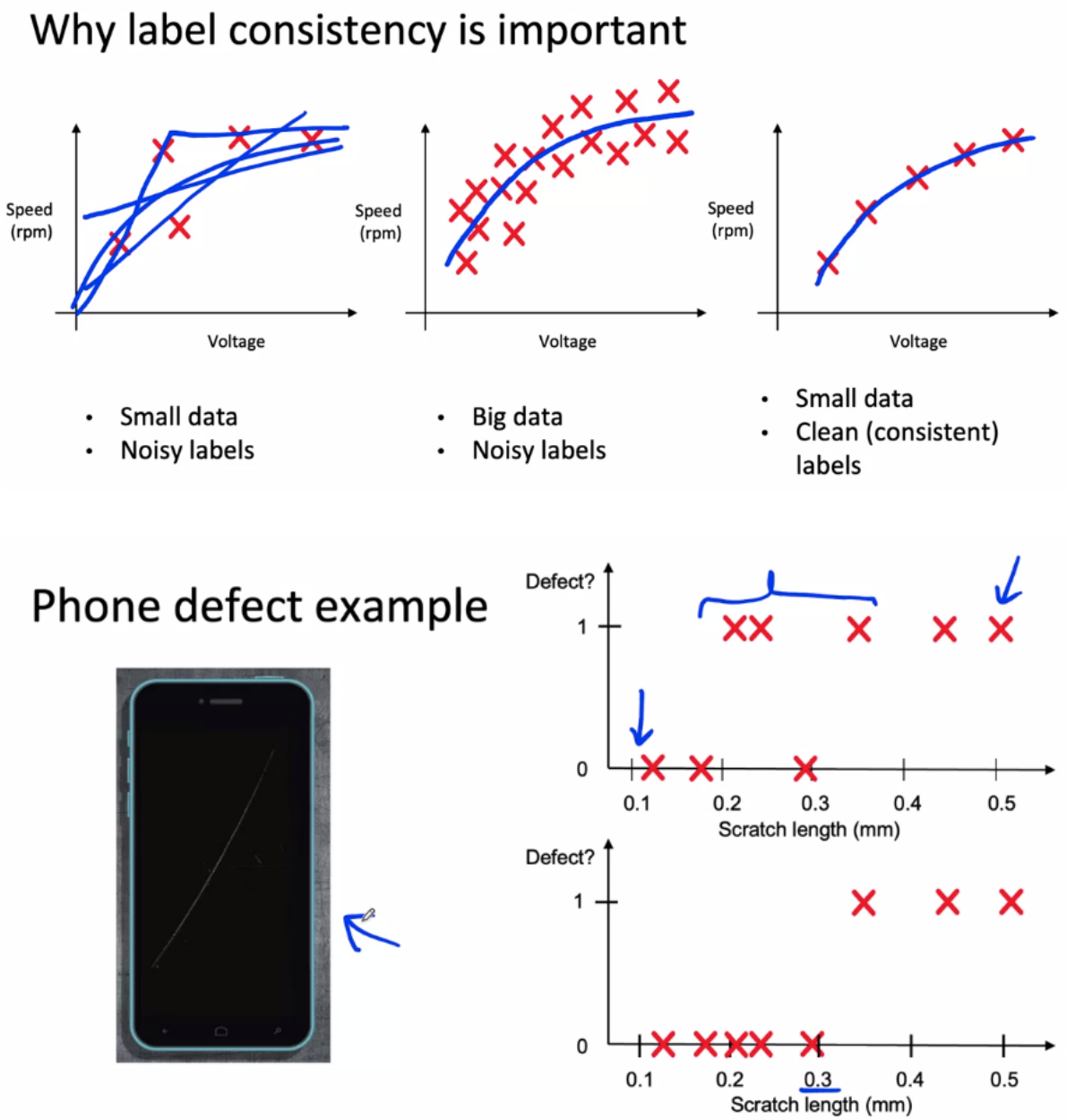


Small data and label consistency

In problems of a small dataset. Having clean and consistent labels is especially important. Let's start with an example. One of the things I used to do is use machine learning to fly helicopters. One thing you might want to do is take the input voltage applied to the motor or to the helicopter rotor and predict what's the speed of the rotor. You can have this type of problem, not just to find helicopters before other control problems with controlling the speed of the motor.

So a pretty small data set because this data set that is the output Y is pretty noisy, it is difficult to know what is the function. It's difficult to know what is the function you should use to map voltage to the rotor speed in rpm. Maybe it should be a straight line.

Now, if you had a ton of data, this data set is equally noisy as the one on the left, but you just have a lot more data. Then the learning algorithm can average over the noisy data sets and you can now fill a function.



So one solution to this would be to say, why don't we try to get a lot more pictures of phones and scratches. And then see what the inspectors do and then maybe eventually we can train a neural network. They can figure out from the image what is and what isn't a scratch on average. Maybe that approach could work, but it'd be a lot of work and require collecting a lot of images. I found that it can be more fruitful to ask the inspectors to sit down and just try to reach agreement on what is the size of scratch. That would cause them to label a scratcher of a bounding box versus decide is too small and not worth bothering labeling.

Big data problems can have small data challenges too

Problems with a large dataset but where there's a long tail of rare events in the input will have small data challenges too.

- Web search
- Self-driving cars
- Product recommendation systems

Improving label consistency

Improving label consistency

- Have multiple labelers label same example.
- When there is disagreement, have MLE, subject matter expert (SME) and/or labelers discuss definition of y to reach agreement.
- If labelers believe that x doesn't contain enough information, consider changing x.
- Iterate until it is hard to significantly increase agreement.

Examples

- Standardize labels

  "Um, nearest gas station"
  "Umm, nearest gas station"
  "Nearest gas station [unintelligible]"

  ➡ "Um, nearest gas station"

- Merge classes

Deep scratch

Shallow scratch

➡ Scratch

Have a class/label to capture uncertainty

- Defect: 0 or 1

Alternative: 0, Borderline, 1

- Unintelligible audio

"nearest go"
"nearest grocery"

"nearest [unintelligible]"

Maybe everyone agrees that the giant scratch is a defect, a tiny scratch is not a defect, but they don't agree on what's in between. If it was possible to get them to agree, then that would be one way to reduce label ambiguity. But if that turns out to be difficult, then here's another option; which is to create a new class where you now have three labels. You can say, it's clearly not a defect, or clearly a defect, or just acknowledge there's some examples that are ambiguous and put them in a new borderline class. If it becomes easier to come up with consistent instructions for this three class problem, because maybe some examples are genuinely borderline, then that could potentially improve labeling consistency.

Small data vs. big data (unstructured data)

Small data

- Usually small number of labelers.
- Can ask labelers to discuss specific labels.

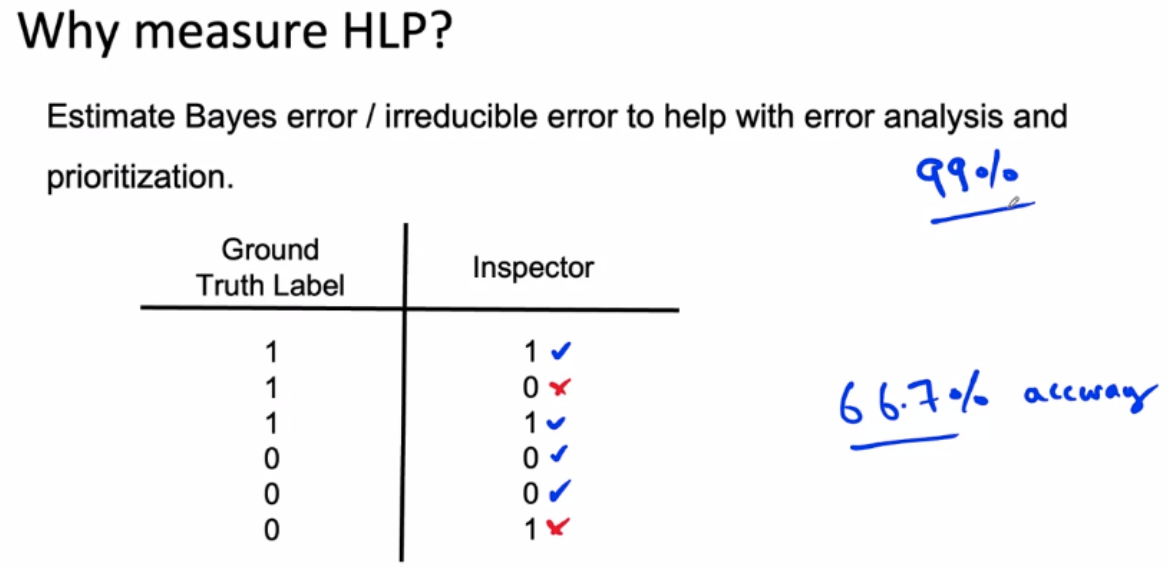
Big data

- Get to consistent definition with a small group.
- Then send labeling instructions to labelers.
- Can consider having multiple labelers label every example and using voting or consensus labels to increase accuracy.

Human level performance (HLP)

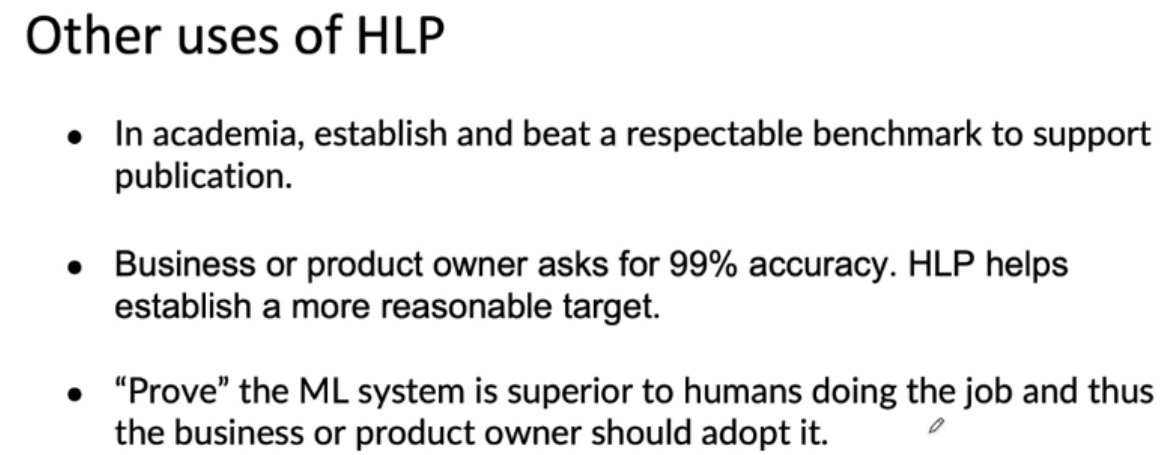
Human Level Performance is also sometimes misuse.

One of the most important users of measuring Human Level Performance or HLP is to estimate based error or irreducible error. Especially on unstructured data tasks in order to help with their analysis and prioritization and just establish what might be possible.



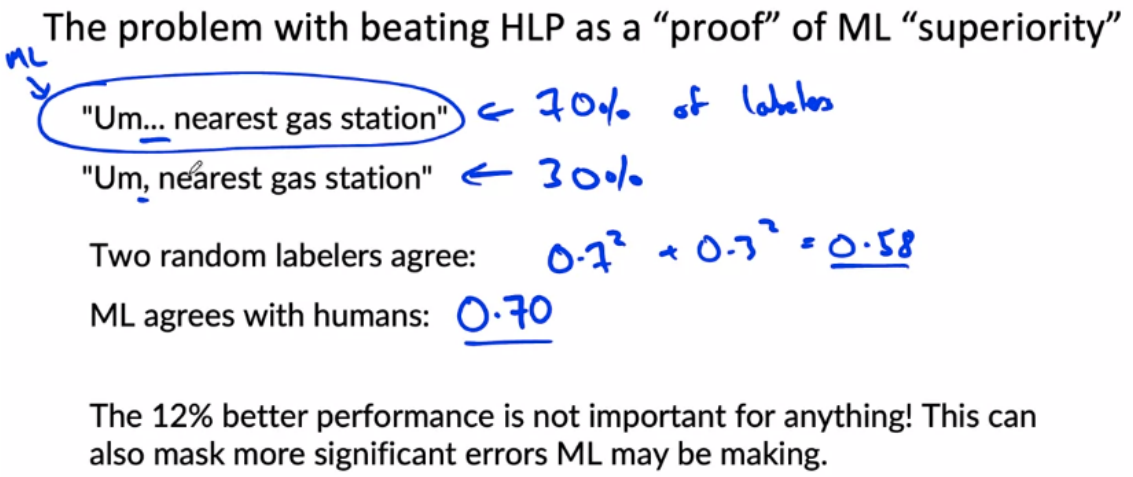
And so this would let you go back to the business owner and say look, even your inspector is only 66.7% accuracy. How can you expect me to Get 99% accuracy?

There's one question that is often not asked, which is what exactly is this Ground Truth Label? Because rather than just measuring how well we can do compared to some Ground Truth Label, which was probably written by some other human. Are we really measuring what is possible or are we just measuring how well two different people happen to agree with each other? When the Ground Truth Label is itself determined by a person. There's a very different approach to thinking about Human Level Performance.

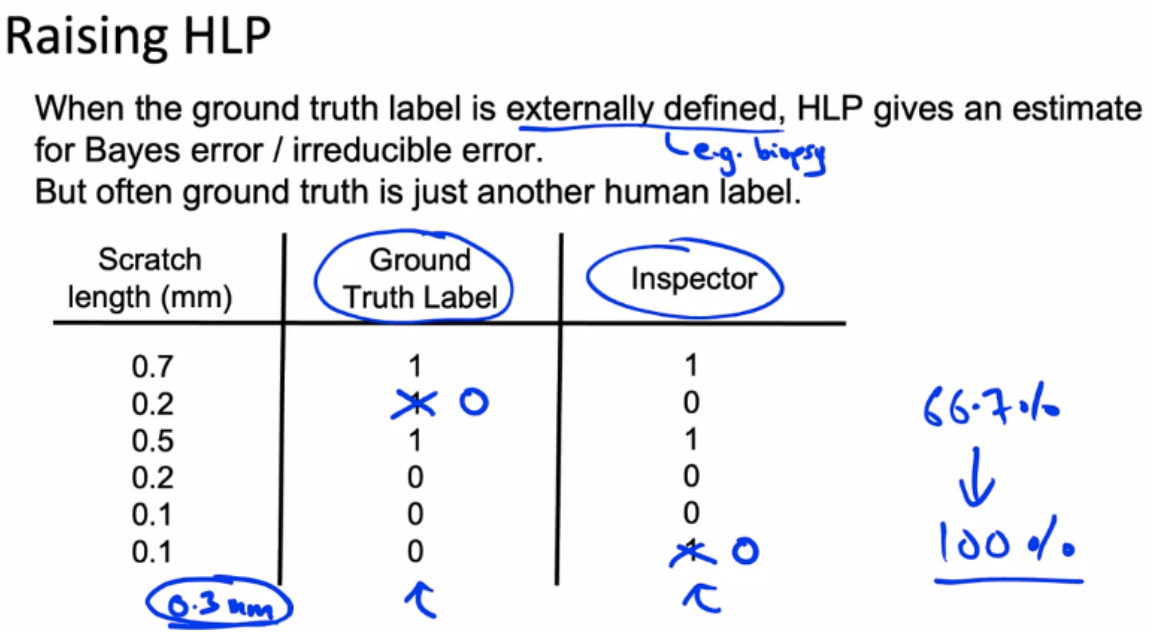


As tempting as it is to go to someone and say look, I've proved that my machinery system is more accurate than humans inspecting the phones or the radiologist reading X-rays or something. And now that I've mathematically proved the superiority of my learning album, you have to use it right? I know the logic of that is tempting, but as a practical matter, this approach rarely works. And you also saw last week that businesses need systems that do more than just doing well on average test set accuracy. So if you ever find yourself in this situation, I would urge

you to just use this type of logic with caution or maybe even more preferably just don't use these arguments.



Raising HLP



Rather than just aspiring to beat the human inspector, it may be more useful to see why the ground truth, which is just some other inspector compared to this inspector don't agree.

have them agree that 0.3 mm is the threshold above which a stretch becomes a defect, then what we realize is that for the first example, both label that one totally appropriately. For the second example, the ground truth here is one but is less than 0.3, so we really should change this to zero,

If we go through this exercise of getting the ground truth label and this inspector to agree, then we actually just raise human-level performance from 66.7 percent to 100 percent, at least as

measured on these six examples. But notice what we've done, by raising HLP to 100 percent we've made it pretty much impossible for learning algorithm to beat HLP, so that seems terrible. You can't tell the business owner anymore, you beat HLP, and thus they must use your system. But the benefit of this is you now have much cleaner, more consistent data, and that ultimately will allow your learning algorithm to do better. When you go is to come up with a learning algorithm that actually generates accurate predictions rather than just proof for some reason that you can beat HLP. I find this approach of working to raise HLP to be more useful.

HLP on structured data

Structured data problems are less likely to involve human labelers, thus HLP is less frequently used.

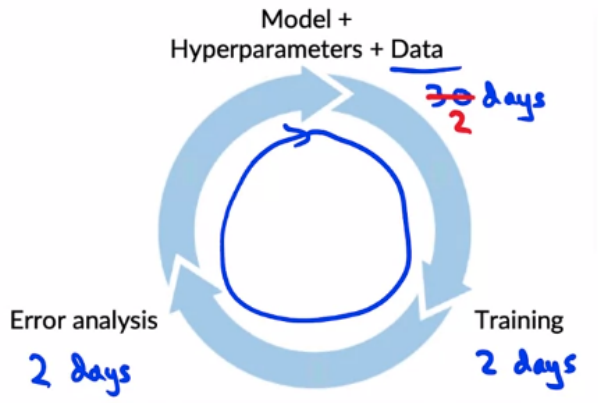
Some exceptions:

- User ID merging: Same person?
- Based on network traffic, is the computer hacked?
- Is the transaction fraudulent?
- Spam account? Bot?
- From GPS, what is the mode of transportation – on foot, bike, car, bus?

Label and Organize Data

Obtaining data

How long should you spend obtaining data?



- Get into this iteration loop as quickly possible.
- Instead of asking: How long it would take to obtain m examples?
  Ask: How much data can we obtain in k days.
- Exception: If you have worked on the problem before and from experience you know you need m examples.

How much time should you spend obtaining data? You know that machine learning is a highly iterative process where you need to pick a model, hyperparameters, have a data set, then training to carry out our analysis and go around this loop multiple times to get to a good model.

"Hey everyone, we're going to spend at most seven days collecting data, so what can we do?" I found that posing the question that way often leads to much more creative ways.

One exception to this guideline is if you have worked on this problem before, and if from experience you know you need at least a certain training set size. Then it might be okay to invest more effort up front to collect that much data.

In terms of getting the data you need, one other step I often carry out is to take inventory of possible data sources.

Inventory data

Brainstorm list of data sources (speech recognition)

Source	Amount	Cost	Time
Owned	100h	$0	0
Crowdsourced – Reading	1000h	$10000	14 days
Pay for labels	100h	$6000	7 days
Purchase data	1000h	$10000	1 day

The three most common ways to get data labeled are, in house where you have your own team label the data, versus outsource where you might find some company that labels data and have them do it for you, versus crowdsource where you might use a crowdsourcing platform to have a large group collectively label the data. The difference between outsource versus crowdsource is that, depending on what type of data you have, there may be specialized companies that could help you get the label quite efficiently.

Some of the trade offs between these options, having machine learning engineers label data is often expensive. But I find that to get a project going quickly, having machine learning engineers do this just for a few days is usually fine and in fact, this can help build the machine learning engineers intuition about the data.

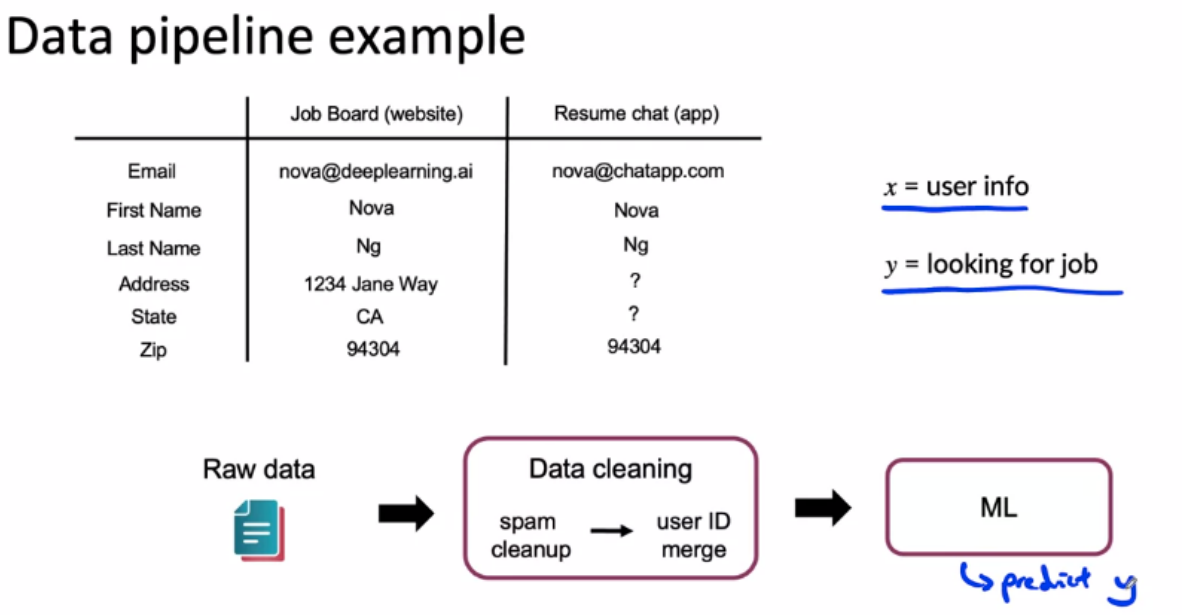
Labeling data

- Options: In-house vs. outsourced vs. crowdsourced
- Having MLEs label data is expensive. But doing this for just a few days is usually fine.
- Who is qualified to label?
  - Speech recognition – any reasonably fluent speaker
  - Factory inspection, medical image diagnosis – SME (subject matter expert)
  - Recommender systems – maybe impossible to label well
- Don't increase data by more than 10x at a time

I found this really hard to predict what will happen when your data set size increases even beyond that. Is also fine to increase your dataset size 10 percent or 50 percent or just 2x at a time, so this is only an upper bound for how much you might invest to increase your dataset size. This guideline hopefully will help teams avoid over investing in tons of data, only to realize that collecting quite that much data wasn't the most useful thing they could have done.

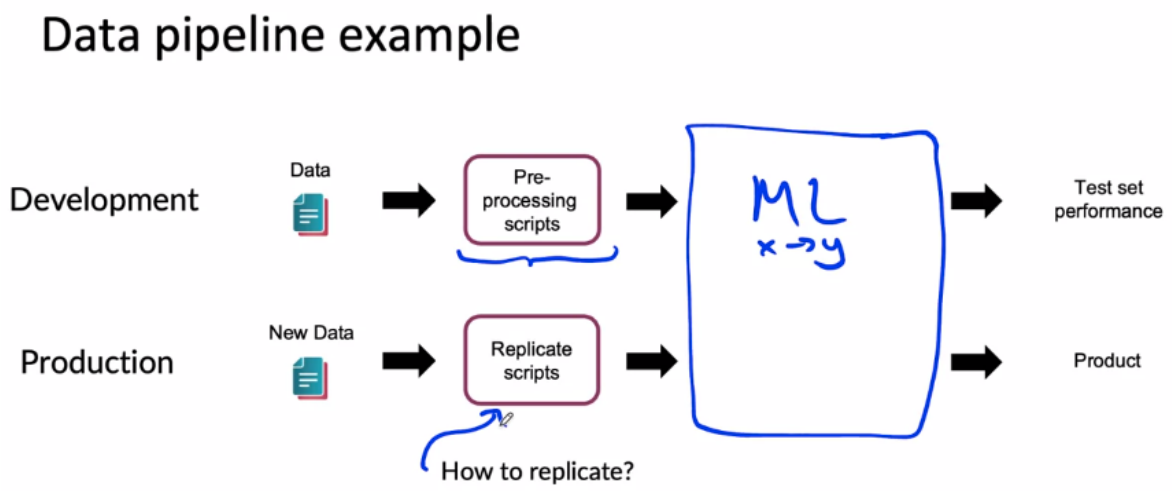
Data pipeline

Data pipelines, sometimes also called Data Cascades refers to when your data has multiple steps of processing before getting to the final output.



How do you replicate the strips to make sure that the input distribution to a machine learning algorithm was the same for the development data and the production data? I find that the amount of effort that you should invest to make sure that the pre-processing scripts are highly replicable

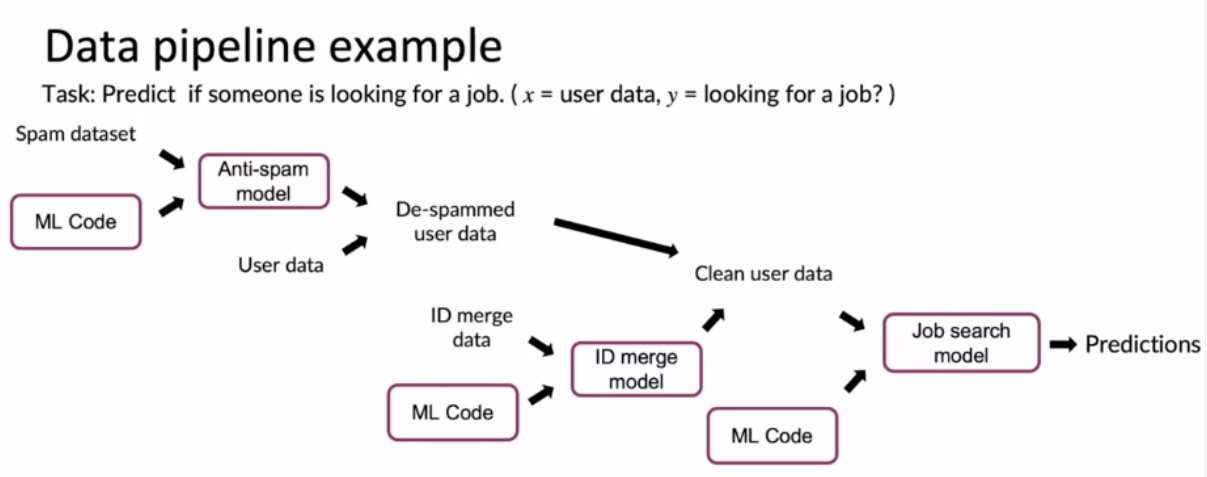
can depend a little bit on the face of the project. I know that it may be fashionable to say that everything you do should be 100 percent replicable



A lot of projects do go through a proof of concept or POC phase, and then a production phase where during the **proof of concept** phase, the primary goal is just to decide if the application is workable and worth building and deploying.

Once you decide that this project is worth taking to production, then you know it's going to be really important to replicate any preprocessing scripts. In this phase, that's when I would use more sophisticated tools to make sure the entire data pipeline is replicable. This is when tools which can be a little bit more heavyweight, but tools like TensorFlow Transform, Apache beam, Airflow, and so on become very valuable.

Meta-data, data provenance and lineage



What if after running this system for months, you discover that, oops, the IP address blacklists you're using have some mistakes in it. In particular, what if you discover that there were some IP addresses that were incorrectly blacklisted.

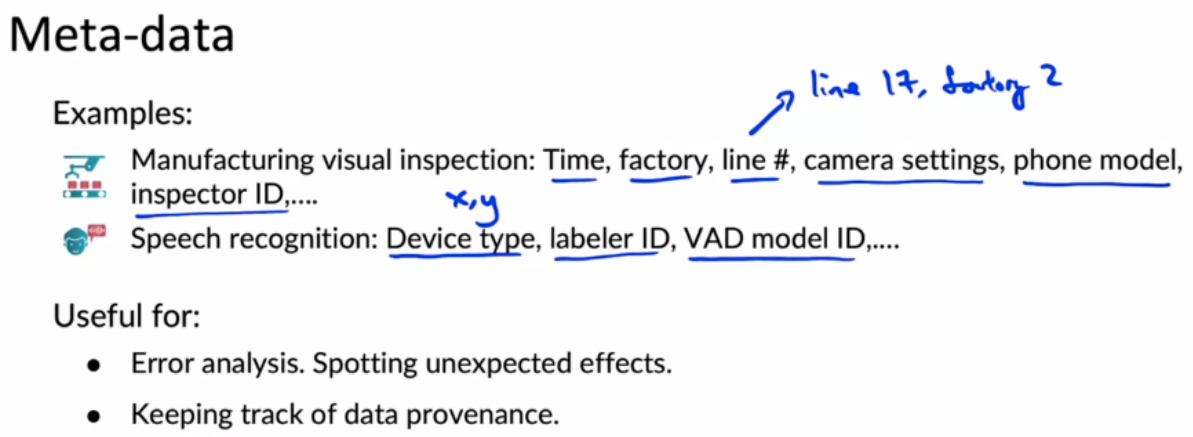
How do you go back and fix this problem? Especially if each of these systems was developed by different engineers, and you have files spread across the laptops of your machine learning engineering development team. To make sure your system is maintainable, especially when a piece of data upstream ends up needing to be changed, it can be very helpful to keep track of data provenance as well as lineage. **Data provenance** refers to where the data came from. Who did you purchase the spam IP address from? **Lineage** refers to the sequence of steps needed to get to the end of the pipeline.

To be honest, the tools for keeping track of data provenance and lineage are still immature into this machine learning world. I find that extensive documentation can help and some formal tools like TensorFlow Transform can also help but solving this type of problem is still not something that we are great at as a community yet.

Metadata is data about data. For example, in manufacturing visual inspection, the data would be the pictures of phones and the labels but if you have metadata that tells you what time was this picture of a phone taken, what factory was this picture from, what's the line number, what were the camera settings such as camera exposure time and camera aperture, what's the number of the phone you're inspecting, what's the ID of the inspector that provided this label. These are examples of data about your dataset X and Y. This type of metadata can turn out to be really useful because if you discover, during machine learning development, that for some strange reason, line number 17 in factory 2 generates images that produce a lot more errors for some reason. Then this allows you to go back to see what was funny about line 17 and factory 2. But if you had not stored the factory in line number metadata in the first place, then it would have been really difficult to discover this during error analysis.

My tip is if you have a framework or a set of MLOps tools for storing metadata, that will definitely make life easier but even if you don't, just like you rarely regret commenting your code, I think you will really regret storing metadata that could then turn out to be useful later.

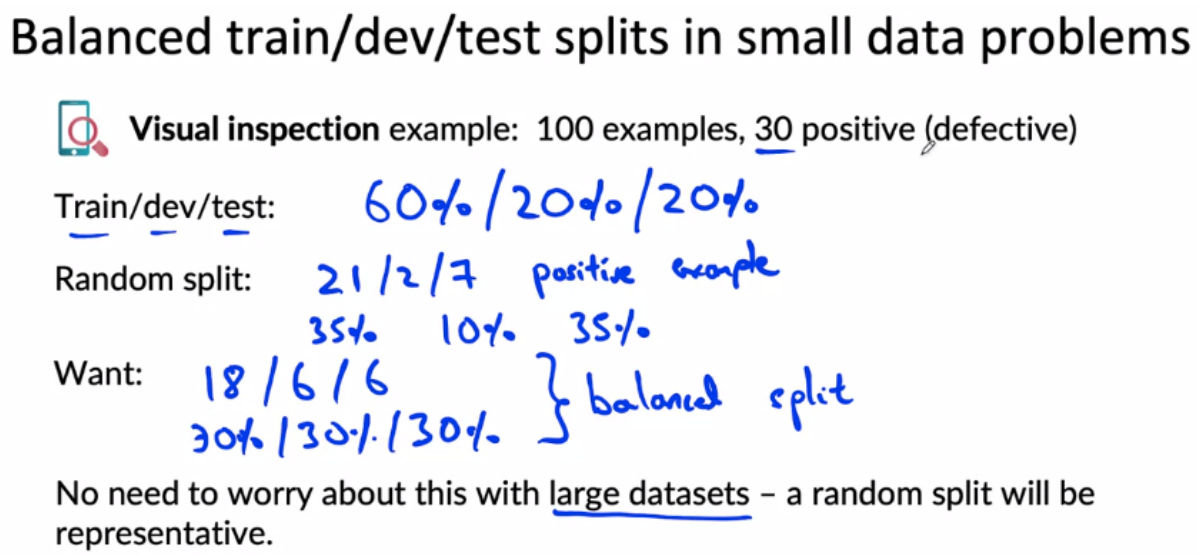
One more example for speech recognition. If you have audio recorded from different brands of smartphones, let's say that in advance, or if you have different labelers labeling your speech, or if you use a voice activity detection model, then just keep track of what was their version number of the voice activity detection model that you use. All of these means that in case for some reason, one version of the VAD, voice activity detection system results in much larger errors, this significantly increases the odds of you discovering that and really use that to improve your learning algorithm performance.



Balanced train/dev/test splits

This makes your dev set quite non representative because in your dev set you have only 2 or 10% positive examples rather than 30% positive examples. But when your data set is small than all of your 20 dev set examples, it's just a higher chance of this, slightly less representative split. So what we would really want is for the training set to have exactly 18 positive examples, dev set to have exactly 6 positive examples and the test set to have exactly 6 positive examples. And this would be 30%, 30% 30%. And if you could get this type of split, this would be called a balanced split.

This makes your data set more representative of the true data distribution.

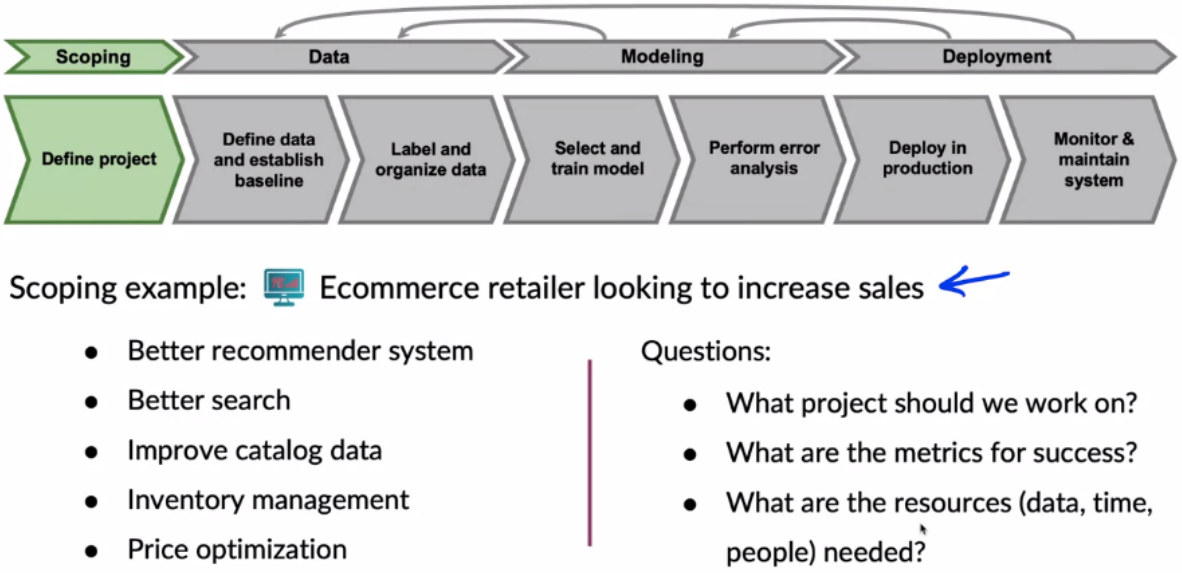


Scoping (optional)

What is scoping?

I find that if you buy yourself or your team is going to spend a lot of time, weeks or months or even longer work on the project is well worth your while to think through a few options and try to select the most promising project to work on before putting so much effort into it.

What I've seen in a lot of businesses is that of all the ideas you can work on, some are going to be much more valuable than others. Maybe two times or five times or 10 times more valuable than a different idea. And being able to pick the most valuable project will significantly increase the impact of your work.



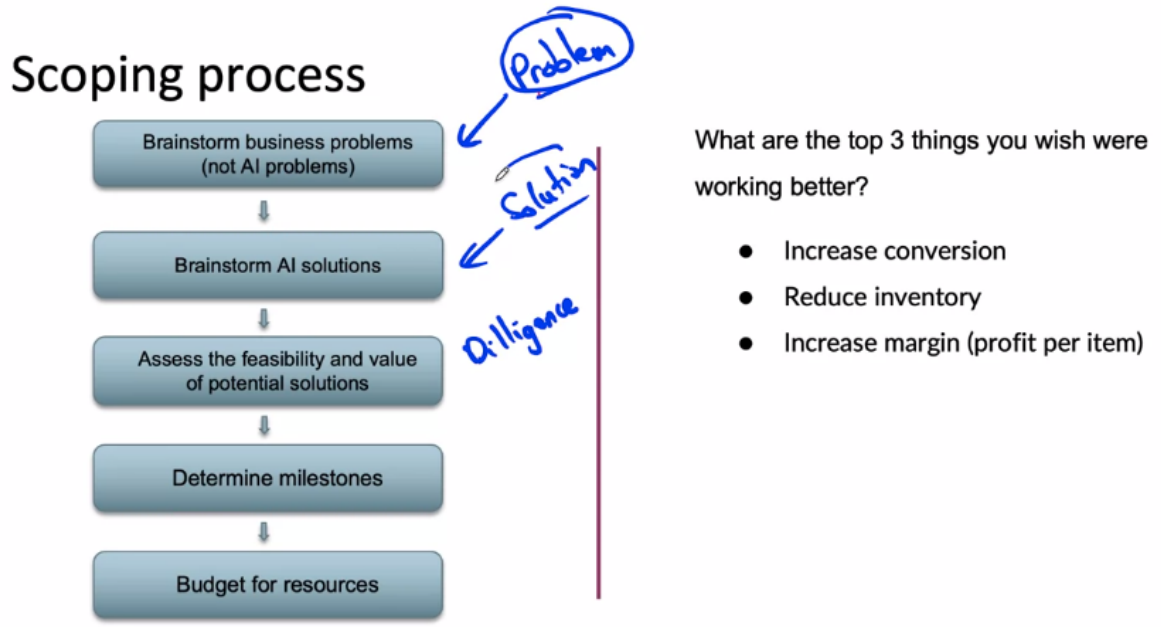
Scoping process

The first thing I do is usually get together with a business or private owner. Often not an Ai person, but someone that understands the business and application and to brainstorm with them. What are their business or application problems? And at this stage I'm trying to identify a business problem, not an Ai problem.

I find it is hopeful for this process to separate out the identification of the problem, from the identification of the solution as engineers. We are pretty good at coming up with solutions, but having a clear articulation of what is the problem first often helps us come up with better solutions.

After brainstorming a variety of different solutions, I would then assess the feasibility and the value of these different solutions. Sometimes you hear me use the word **diligence**, to refer to this phrase.

After validating technical feasibility and value or ROI. Return on investment if you can project if it still looks promising right, if it still looks promising. We then flesh out the milestones for the project and finally budget for resources.



The problem identification is a step of thinking through whether the things you want to achieve. And solution identification is a process of thinking through how to achieve those objectives.

Separating problem identification from solution

Problem	Solution
Increase conversion	Search, recommendations
Reduce inventory	Demand prediction, marketing
Increase margin (profit per item)	Optimizing what to sell (e.g., merchandising), recommend bundles

Diligence on feasibility and value

New means you are delivering a brand new capability and the existing means you're scoping out the project to improve on an existing capability.

When evaluating HLP, I would give a human the same data as would be fed to a learning algorithm and just ask, can a human given the same data, perform the tasks such as can the human given a picture of a scratch smartphone, perform the task of detecting scratches reliably? If a human can do it, then that significantly increases the hope they can also get the learning algorithm to do it. For existing projects, I would use HLP as a reference as well. Where if you have a visual inspection system and you're hoping to improve it to a certain level of performance.

If humans can achieve the level you're hoping to get to, then that might give you more hope that it is technically feasible. Whereas if you're hoping to increase performance well beyond human level performance, then that suggests the project might be harder or may not be possible. In addition to HLP, I often also use the history of the project as a predictor for future progress.

Moving over to the right column. If you're working on a brand new project with structure data, the question I would ask is, are predictive features available? Do you have reason to think that the data you have, the inputs X are strongly predictive or sufficiently predictive of the target outputs Y? In this box on the lower right, for structured data problems, if you're trying to improve an existing system, one thing that will help a lot is if you can identify new predictive features. Are there features that you aren't yet using but you can identify that could really help predict Y and also by looking at? The history of the project.

Feasibility: Is this project technically feasible?

Use external benchmark (literature, other company, competitor)

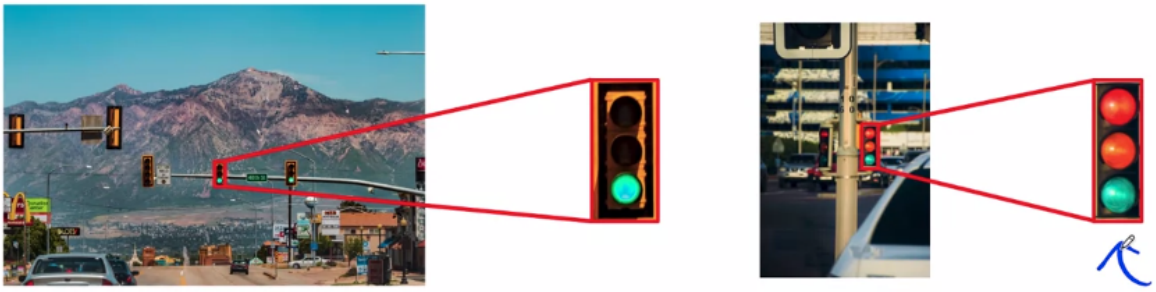
	Unstructured (e.g., speech, images)	Structured (e.g., transaction records)
New	HLP	Predictive features available?
Existing	HLP History of project	New predictive features? History of project

HLP: Can a human, given the same data, perform the task?

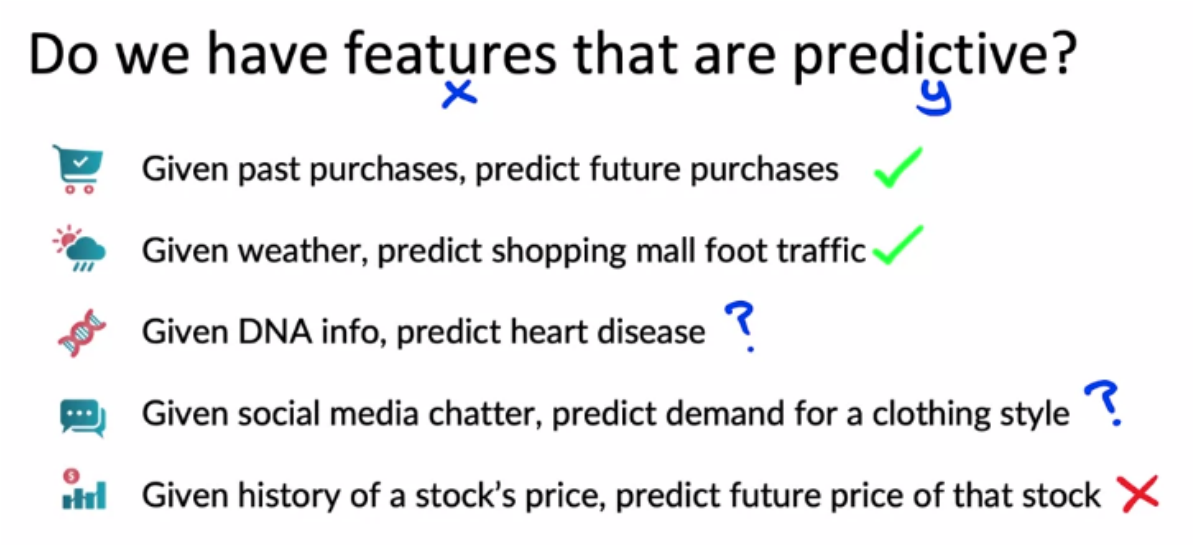
Why use HLP to benchmark?

People are very good on unstructured data tasks

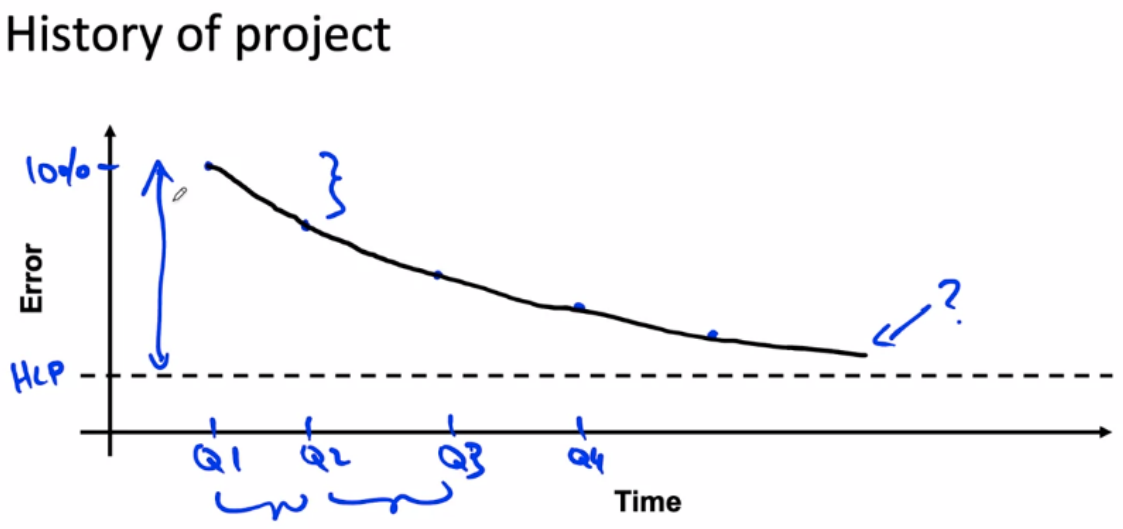
Criteria: Can a human, given the same data, perform the task?



Making sure that a human sees only the same data as a learning algorithm will see is really important. I've seen a lot of projects where for a long time a team was working on a computer vision system, say, and they thought they could do it because a human physically inspecting the cell phone or something could detect the defect. But it took a long time to realize that even a human looking only at the image, couldn't figure out what was going on. If you can realize that earlier, then you can figure much earlier that with the current camera set up, it just wasn't visible. The more efficient thing to do would have been to invest early on in a better camera or better lighting setup or something, rather than keep working on a machine learning algorithm on the problem that I think just wasn't durable, with the imaging setup available at a time.



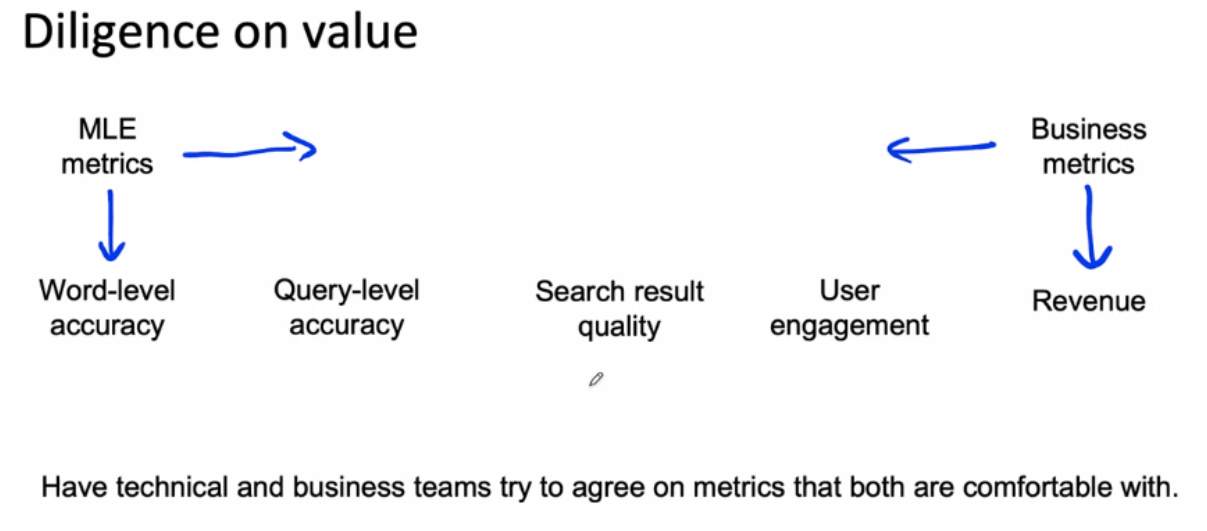
By estimating this rate of progress, you may project into the future that hopefully in future quarters, you continue to reduce the error by 30 percent relative to HRP. This will give you a sense of what might be reasonable for the future rate of progress on this project. This gives you a sense of what may be feasible for an existing project for which you already have this type of history and can try to extrapolate into the future.



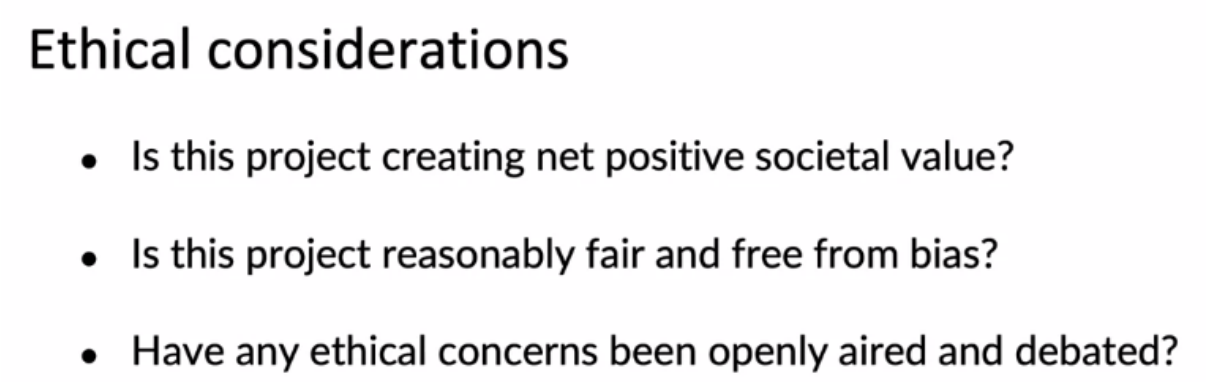
Diligence on value

How do you estimate the value of a Machine Learning Project?

The further we go to the right, the harder it is for a machine learning team to really give a guarantee. I wish more problems could be solved by gradient descent or by optimizing tested accuracy. But that's just not to say of the world today, a lot of practical problems require us to do something more than just optimizing tested accuracy.



One other practice I've found useful is that you can do even very rough back of the envelope calculations to relate what level of accuracy to some of the metrics on the right. If word accuracy improves by one percent, if you have any rough guess for will that improve query level accuracy, maybe by 0.7 percent or 0.8 percent, and how much will that improve search result quality and user engagement and maybe revenue, if able to come up with even a very crude back of the envelope calculation. Sometimes these are also called **Fermi estimates**.



Milestones and resourcing

Milestones & Resourcing

Key specifications:

- ML metrics (accuracy, precision/recall, etc.)
- Software metrics (latency, throughput, etc. given compute resources)
- Business metrics (revenue, etc.)
- Resources needed (data, personnel, help from other teams)
- Timeline

If unsure, consider benchmarking to other projects, or building a POC (Proof of Concept) first.

Week 3 Optional References

Week 3: Data Definition and Baseline

If you wish to dive more deeply into the topics covered this week, feel free to check out these optional references. You won't have to read these to complete this week's practice quizzes.

Label ambiguity

Data pipelines

Data lineage

MLops

Geirhos, R., Janssen, D. H. J., Schutt, H. H., Rauber, J., Bethge, M., & Wichmann, F. A. (n.d.). Comparing deep neural networks against humans: object recognition when the signal gets weaker∗. Retrieved May 7, 2021, from Arxiv.org website: https://arxiv.org/pdf/1706.06969.pdf

References

Introduction to Machine Learning in Production

This is a compilation of resources including URLs and papers appearing in lecture videos.

Overall resources:

Konstantinos, Katsiapis, Karmarkar, A., Altay, A., Zaks, A., Polyzotis, N., … Li, Z. (2020). Towards ML Engineering: A brief history of TensorFlow Extended (TFX). http://arxiv.org/abs/2010.02013

Paleyes, A., Urma, R.-G., & Lawrence, N. D. (2020). Challenges in deploying machine learning: A survey of case studies. http://arxiv.org/abs/2011.09926

Course #2 Machine Learning Data Lifecycle in Production

Week 1: Collecting, Labeling and Validating Data

This week covers a quick introduction to machine learning production systems. More concretely you will learn about leveraging the TensorFlow Extended (TFX) library to collect, label and validate data to make it production ready.

Learning Objectives

- **Describe the differences between ML modeling and a production ML system**
- **Identify responsible data collection for building a fair production ML system**
- **Discuss data and concept change and how to address it by annotating new training data with direct labeling and/or human labeling**
- **Address training data issues by generating dataset statistics and creating, comparing and updating data schemas**

Week 2: Feature Engineering, Transformation and Selection

Implement feature engineering, transformation, and selection with TensorFlow Extended by encoding structured and unstructured data types and addressing class imbalances

Learning Objectives

- **Define a set of feature engineering techniques, such as scaling and binning**
- **Use TensorFlow Transform for a simple preprocessing and data transformation task**
- **Describe feature space coverage and implement different feature selection methods**
- **Perform feature selection using scikit-learn routines and ensure feature space coverage**

Week 3: Data Journey and Data Storage

Understand the data journey over a production system's lifecycle and leverage ML metadata and enterprise schemas to address quickly evolving data.

Learning Objectives

- **Describe data journey through data lineage and provenance**

- **Integrate the sequence of pipeline artifacts into metadata storage using ML Metadata library**
- **Iteratively create enterprise data schema**
- **Explain how to integrate enterprise data into feature stores, data warehouses and data lakes**

Week 4 (Optional): Advanced Labeling, Augmentation and Data Preprocessing

Combine labeled and unlabeled data to improve ML model accuracy and augment data to diversify your training set.

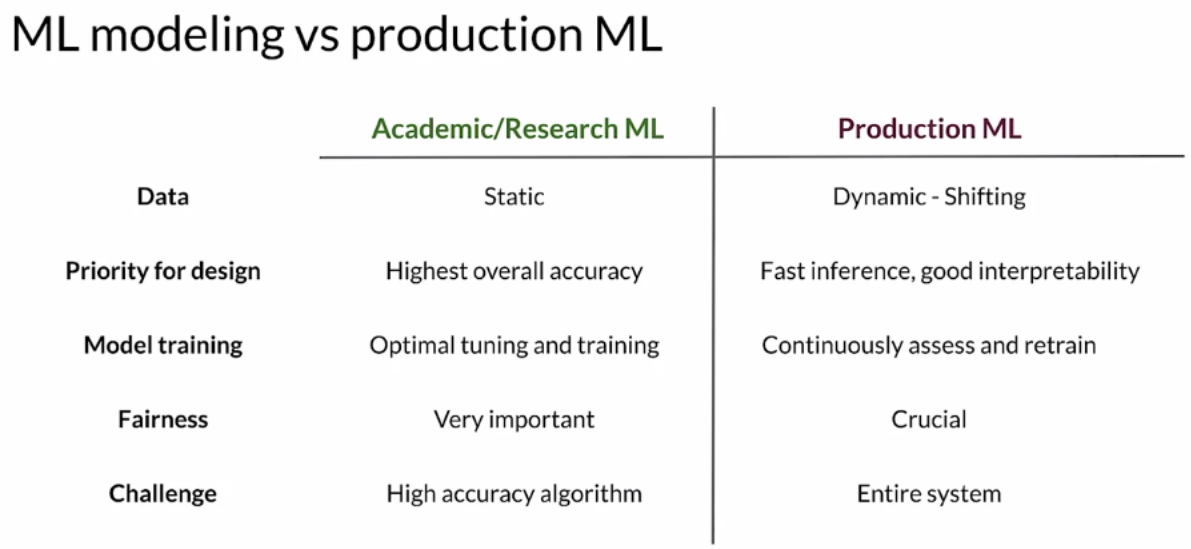
Learning Objectives

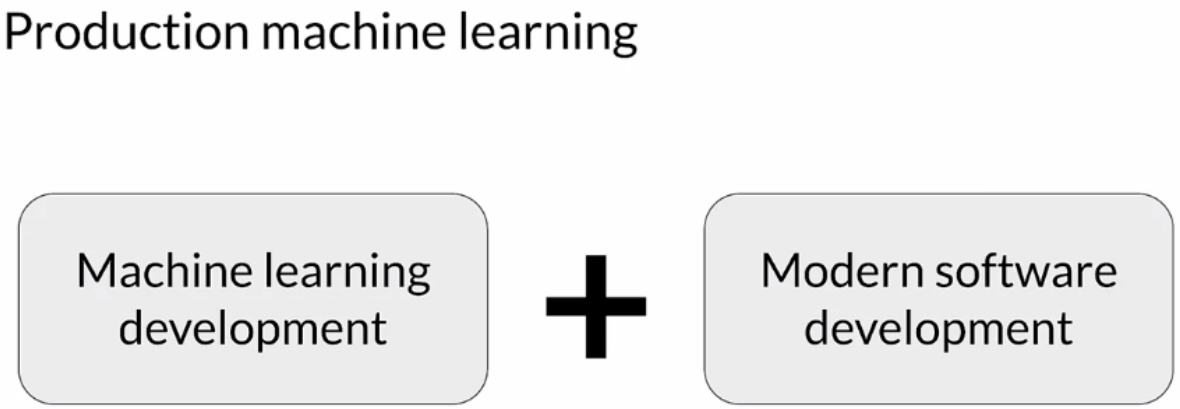
- **Discuss direct, semi-supervised, weak supervision and active learning methods for labeling data**
- **Increase the diversity of your training set by data augmentation**
- **Perform advanced data preparation and transformation on different structured and unstructured data types**

Week 1: Collecting, Labeling and Validating Data

Introduction to Machine Learning Engineering in Production

Overview





Managing the entire life cycle of data

- Labeling
- Feature space coverage
- Minimal dimensionality
- Maximum predictive data
- Fairness
- Rare conditions

Modern software development

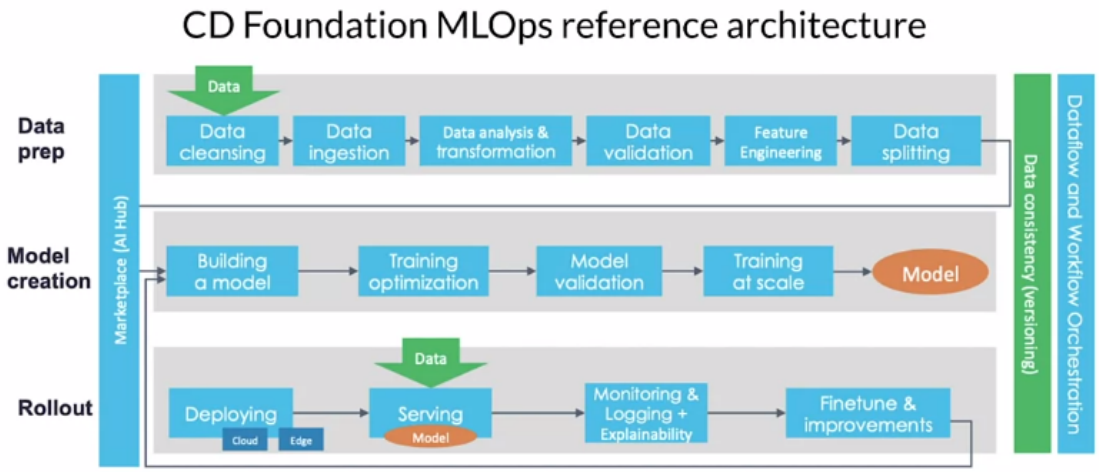
Accounts for:

- Scalability
- Extensibility
- Configuration
- Consistency & reproducibility
- Safety & security
- Modularity
- Testability
- Monitoring
- Best practices

Challenges in production grade ML

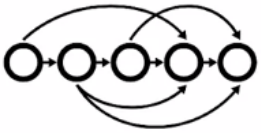
- Build integrated ML systems
- Continuously operate it in production
- Handle continuously changing data
- Optimize compute resource costs

Production ML infrastructure

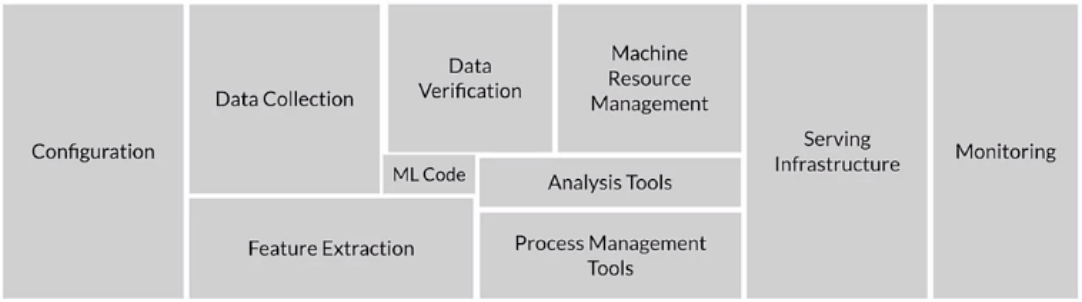


Directed acyclic graphs

- A directed acyclic graph (DAG) is a directed graph that has no cycles
- ML pipeline workflows are usually DAGs
- DAGs define the sequencing of the tasks to be performed, based on their relationships and dependencies.

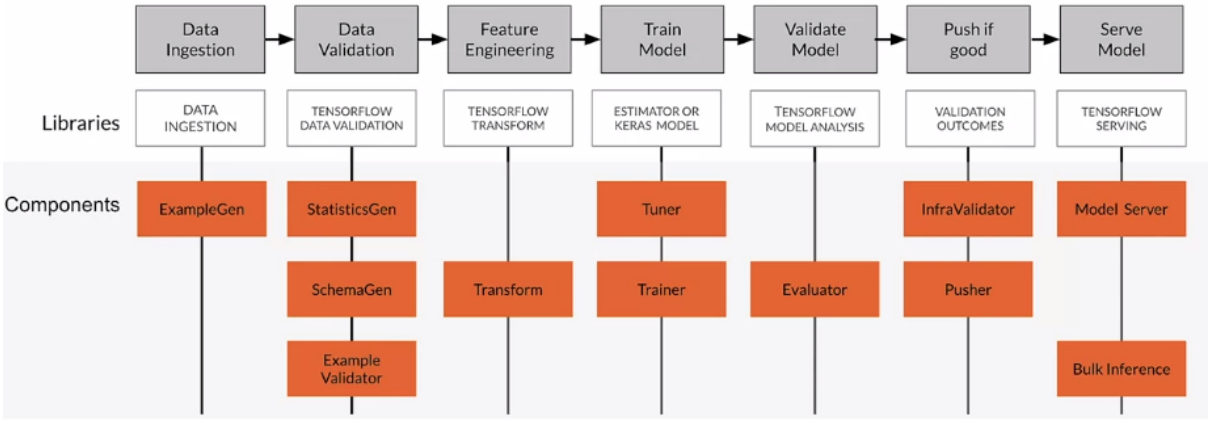


Pipeline orchestration frameworks

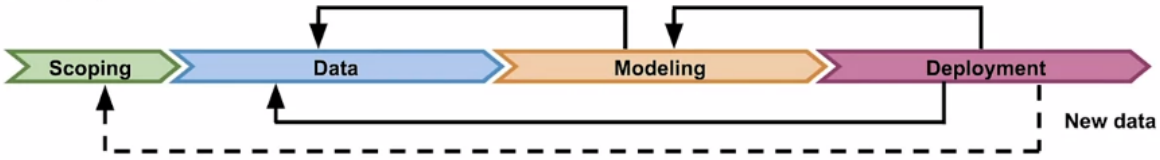


- Responsible for scheduling the various components in an ML pipeline DAG dependencies
- Help with pipeline automation
- Examples: Airflow, Argo, Celery, Luigi, Kubeflow

TFX production components



Key points



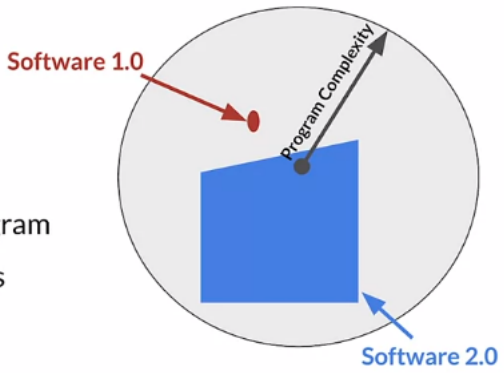
- Production ML pipelines: automating, monitoring, and maintaining end-to-end processes
- Production ML is much more than just ML code
  - ML development + software development
- TFX is an open-source end-to-end ML platform

Collecting Data

Importance of Data

ML: Data is a first class citizen

- Software 1.0
  - Explicit instructions to the computer
- Software 2.0
  - Specify some goal on the behavior of a program
  - Find solution using optimization techniques
  - Good data is key for success
  - Code in Software = Data in ML



Everything starts with data

- Models aren't magic
- Meaningful data:
  - maximize predictive content
  - remove non-informative data
  - feature space coverage

Garbage in, garbage out

$$f(\text{🗑}) = \text{🗑}$$

Key Points

- Understand users, translate user needs into data problems
- Ensure data coverage and high predictive signal
- Source, store and monitor quality data responsibly

Example Application: Suggesting Runs

Example application: Suggesting runs

Users	Runners
User Need	Run more often
User Actions	Complete run using the app
ML System Output	• What routes to suggest • When to suggest them
ML System Learning	• Patterns of behaviour around accepting run prompts • Completing runs • Improving consistency

Key considerations

- Data availability and collection
  - What kind of/how much data is available?
  - How often does the new data come in?
  - Is it annotated?
    - If not, how hard/expensive is it to get it labeled?
- Translate user needs into data needs
  - Data needed
  - Features needed
  - Labels needed

Example dataset

		FEATURES				
	Runner ID	**Run**	**Runner Time**	**Elevation**	**Fun**	
EXAMPLES	AV3DE	Boston Marathon	03:40:32	1,300 ft	Low	LABELS
	X8KGF	Seattle Oktoberfest 5k	00:35:40	0 ft	High	
	BH9IU	Houston Half-marathon	02:01:18	200 ft	Medium	

Get to know your data

- Identify data sources
- Check if they are refreshed
- Consistency for values, units, & data types
- Monitor outliers and errors

Translate user needs into data needs

Data Needed	• Running data from the app • Demographic data • Local geographic data

Features Needed	• Runner demographics • Time of day • Run completion rate • Pace • Distance ran • Elevation gained • Heart rate

Labels Needed	• Runner acceptance or rejection of app suggestions • User generated feedback regarding why suggestion was rejected • User rating of enjoyment of recommended runs

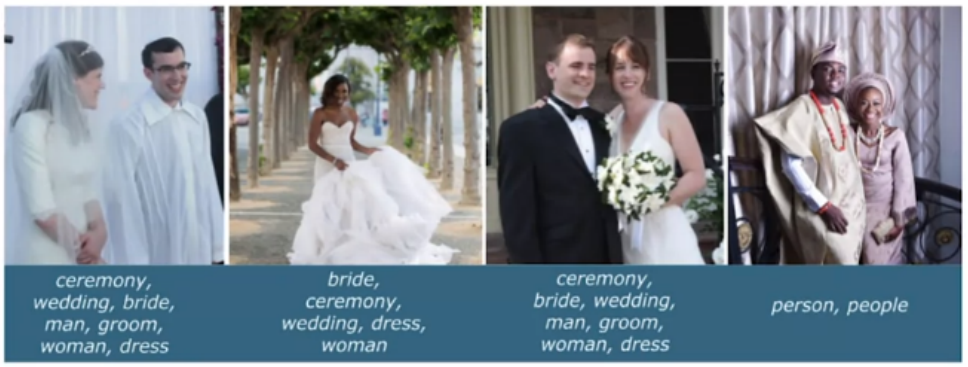
Key points

- Understand your user, translate their needs into data problems
  - What kind of/how much data is available
  - What are the details and issues of your data
  - What are your predictive features
  - What are the labels you are tracking
  - What are your metrics

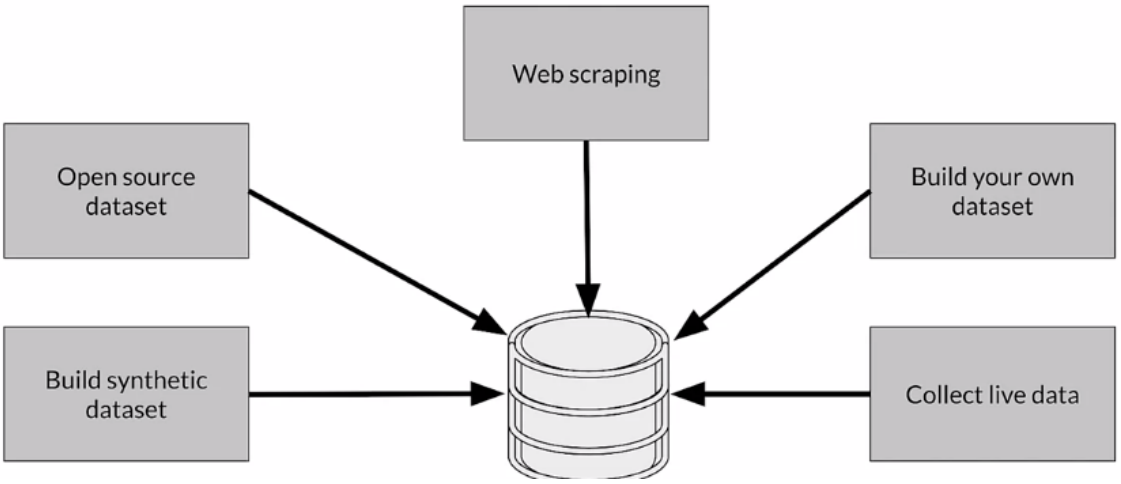
Responsible Data: Security, Privacy & Fairness

Avoiding problematic biases in datasets

Example: classifier trained on the Open Images dataset



Source Data Responsibly



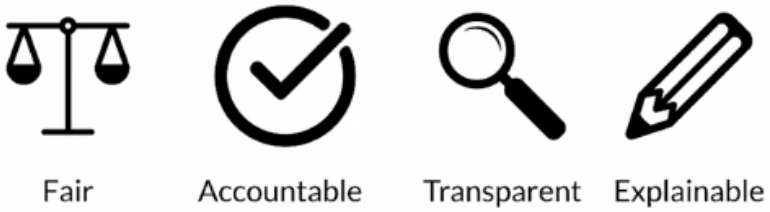
Data security and privacy

- Data collection and management isn't just about your model
  - Give user control of what data can be collected
  - Is there a risk of inadvertently revealing user data?
- Compliance with regulations and policies (e.g. GDPR)

Users privacy

- Protect personally identifiable information
  - Aggregation - replace unique values with summary value
  - Redaction - remove some data to create less complete picture

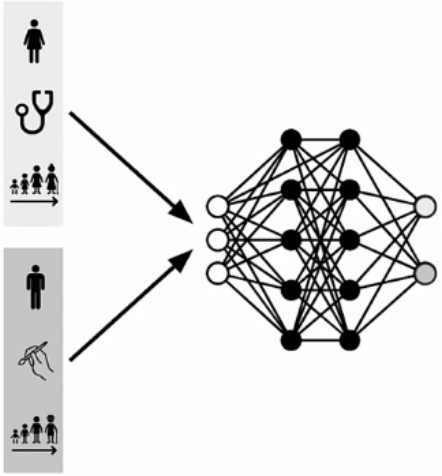
How ML systems can fail users



- Representational harm
- Opportunity denial
- Disproportionate product failure
- Harm by disadvantage

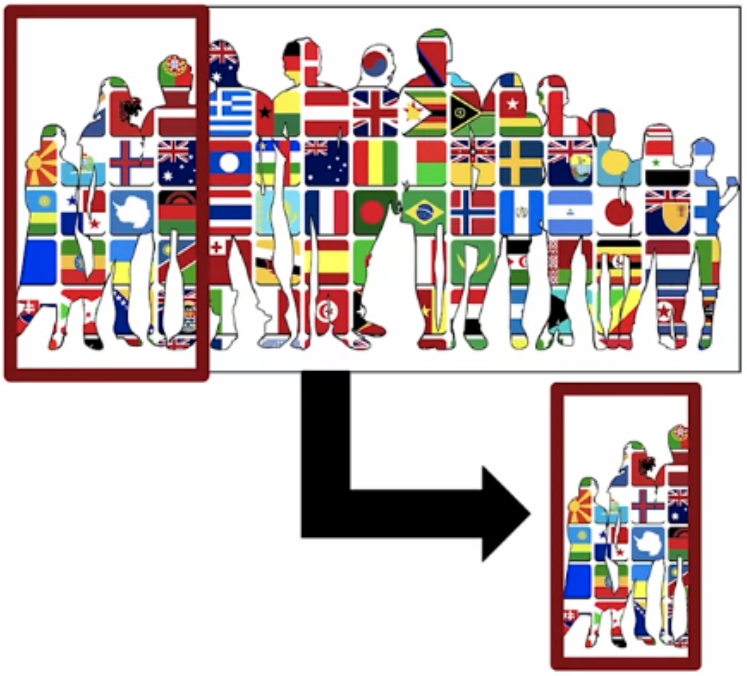
Some of the ways that ML Systems can fail are through things like representational harm. So **representational harm** is where a system will amplify or reflect a negative stereotype about particular groups. **Opportunity denial** is when a system makes predictions that have negative real life consequences that could result in lasting impacts. **Disproportionate product failure** is where the effectiveness of your model is really skewed so that the outputs happen more frequently for particular groups of users, you get skewed outputs more frequently essentially can think of as errors more frequently. **Harm by disadvantage** is where a system will infer disadvantageous associations between different demographic characteristics and the user behaviors around that.

Commit to fairness



- Make sure your models are fair
  - Group fairness, equal accuracy
- Bias in human labeled and/or collected data
- ML Models can amplify biases

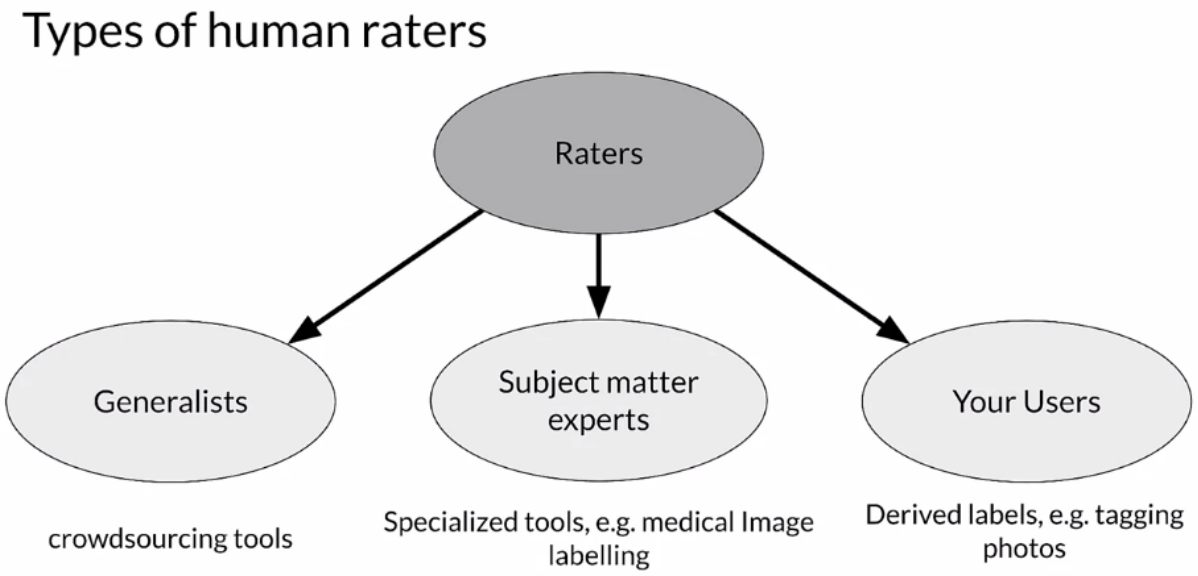
Biased data representation

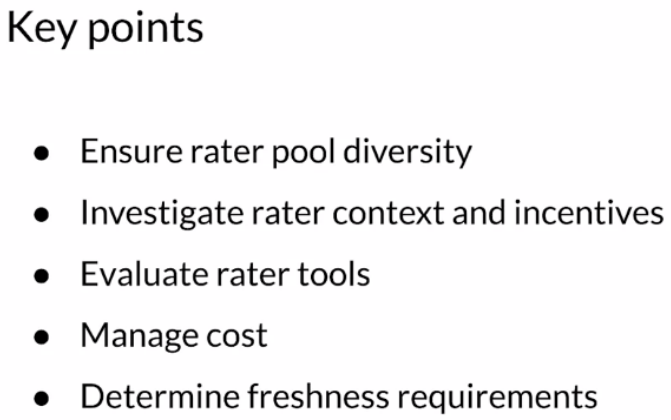


Reducing bias: Design fair labeling systems

- Accurate labels are necessary for supervised learning

- Labeling can be done by:
  - Automation (logging or weak supervision)
  - Humans (aka “Raters”, often semi-supervised)





Labeling Data

Case Study: Degraded Model Performance

To illustrate some of the issues with labeled data in performance settings, let's take a look at a case study.

So imagine that you're an online retailer and you're selling shoes and you have a model that predicts click through rates which helps you to decide how much inventory to order.

Why? You trained your model? You did fairly well in your metrics and so forth. Why is it different? What changed? Why does your model not predict men's dress shoes well now? And perhaps more importantly, how do you even know that you have a problem?

if you don't put good practices into place in the production setting, you're probably going to find out either when you order way too many shoes or not enough shoes. And that's not a situation that you want to be in in a business. This is going to cost you money.

Case study: taking action

- How to detect problems early on?
- What are the possible causes?
- What can be done to solve these?

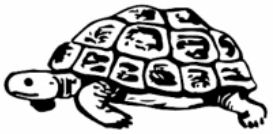
What causes problems?

Kinds of problems:

- Slow - example: drift
- Fast - example: bad sensor, bad software update

Gradual problems

Data changes	• Trend and seasonality • Distribution of features changes • Relative importance of features changes
World changes	• Styles change • Scope and processes change • Competitors change • Business expands to other geos



So, maybe last year black shoes were really fashionable for men's dress shoes and now it's brown shoes or what have you.

Sudden problems

Data collection problem	Systems problem
• Bad sensor/camera • Bad log data • Moved or disabled sensors/cameras	• Bad software update • Loss of network connectivity • System down • Bad credentials

Why “Understand” the model?

- Mispredictions do not have uniform **cost** to your business
- The **data you have** is rarely the data you wish you had
- Model objective is nearly always a **proxy** for your business objectives
- Some percentage of your customers may have a **bad experience**

The real world does not stand still!

In the case of the shoes that we just looked at, we were predicting click through rates as a proxy for deciding how much inventory to order.

Data and Concept Change in Production ML

Detecting problems with deployed models

- Data and scope changes
- Monitor models and validate data to find problems early
- Changing ground truth: **label** new training data

Easy problems

- Ground truth changes slowly (months, years)
- Model retraining driven by:
  - Model improvements, better data
  - Changes in software and/or systems
- Labeling
  - Curated datasets
  - Crowd-based

Harder problems

- Ground truth changes faster (weeks)
- Model retraining driven by:
  - **Declining model performance**
  - Model improvements, better data
  - Changes in software and/or system
- Labeling
  - Direct feedback
  - Crowd-based

Really hard problems

- Ground truth changes very fast (days, hours, min)
- Model retraining driven by:
  - **Declining model performance**
  - Model improvements, better data
  - Changes in software and/or system
- Labeling
  - Direct feedback
  - Weak supervision

Key points

- Model performance decays over time
  - Data and Concept Drift
- Model retraining helps to improve performance
  - Data labeling for changing ground truth and scarce labels

Data labeling

Variety of Methods

- Process Feedback (Direct Labeling)
- Human Labeling
- ~~Semi-Supervised Labeling~~
- ~~Active Learning~~
- ~~Weak Supervision~~

➡ Practice later as advanced labeling methods

Data labeling

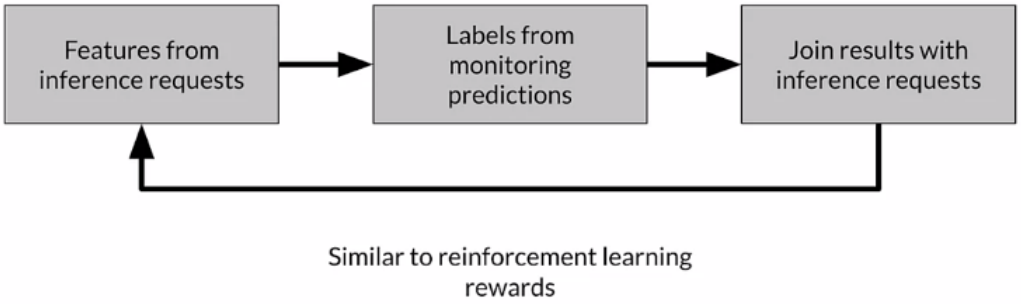
Process Feedback	Example: Actual vs predicted click-through
Human Labeling	Example: Cardiologists labeling MRI images

Actual versus predicted click-through rates. Suppose you have recommendations that you are giving to a user, did they actually click on the things that you recommend? If they did, you can label it positive, if they didn't you can label it negative. Human labeling, you can have humans look at data and apply labels to them. For example, you can ask cardiologists to look at MRI images and apply labels to them.

Why is labeling important in production ML?

- Using business/organisation available data
- Frequent model retraining
- Labeling ongoing and critical process
- Creating a training datasets requires labels

Direct labeling: continuous creation of training dataset



Labeling is an ongoing and often critical process in your application and your business. But at the end of the day, creating a training data set for supervised learning requires labels, you need to think about how you're going to do that. **Direct labeling**, which we'll talk about first or process feedback, is a way of continuously creating new training data that you're going to use to retrain your model. You're taking the features themselves from the inference requests that your model is getting. The predictions that your model is being asked to make and the features that are provided for that. You get labels for those inference requests by monitoring systems and using the feedback from those systems to label that data. One of the things that you need to solve there is to join the results that you get from monitoring those systems with the original inference request which could be hours or days apart. You might've run batches on Monday and you're getting feedback on Friday. You need to make sure that you can do those joins to apply those labels. In some ways you can think about this as similar to reinforcement learning, where instead of applying rewards based on action you're applying labels based on a prediction.

Process feedback - advantages

- Training dataset continuous creation
- Labels evolve quickly
- Captures strong label signals

Process feedback - disadvantages

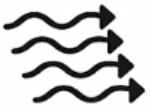
- Hindered by inherent nature of the problem
- Failure to capture ground truth
- Largely bespoke design

Process feedback - Open-Source log analysis tools

Logstash

Free and open source data processing pipeline

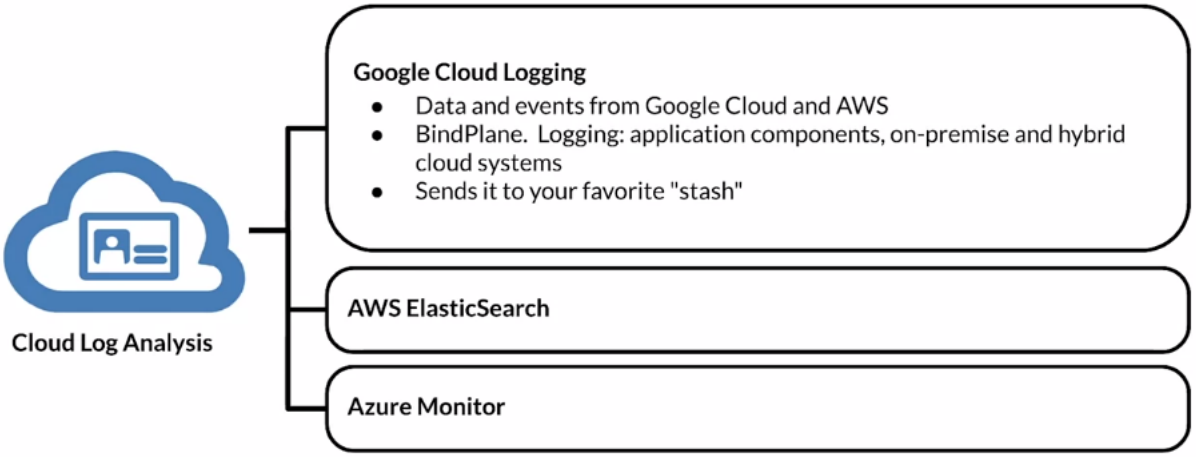
- Ingests data from a multitude of sources
- Transforms it
- Sends it to your favorite "stash."



Fluentd

Open source data collector
Unify the data collection and consumption

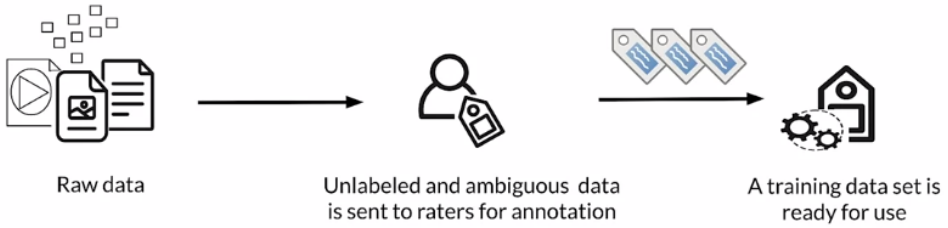
Process feedback - Cloud log analytics



Now, let's turn to human labeling. In human labeling, you have people, humans and we refer to those as raters. We ask them to examine data and assign labels to it. It sounds simple, it sounds like it might be painful but it's the way that a lot of data is generated and labeled. You start with raw data and you give it to people and you ask them to apply labels to it. That's the way that you create a training data set that you're going to use to train or retrain your model.

Human labeling

People (“raters”) to examine data and assign labels manually

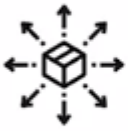


Human labeling - Methodology


Unlabeled data is collected


Human “raters” are recruited

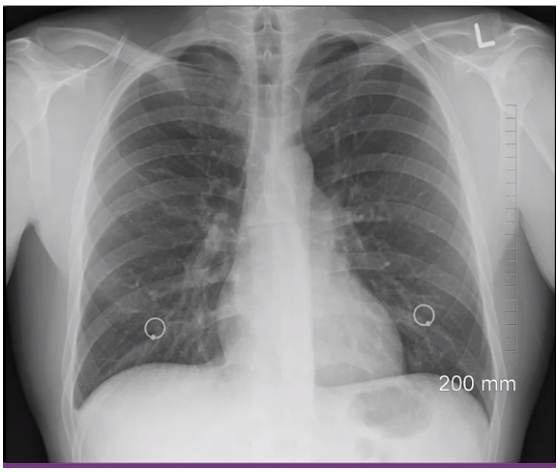

Instructions to guide raters are created


Data is divided and assigned to raters


Labels are collected and conflicts resolved

Human labeling - advantages

- More labels
- Pure supervised learning



One disadvantage, one issue here is that depending on the data that you have it can be very complex for a human to look at it and decide what the label should be. Something like this we might need a radiologist to look at this and tell us what the right labels should be which can be very expensive. But if you're also talking about situations where you have high dimensional data, it's very difficult for humans to look at, say, 100 different features and decide what the label is.

Human labeling - Disadvantages

- Quality consistency: Many datasets difficult for human labeling
- Slow
- Expensive
- Small dataset curation

Why is human labeling a problem?

Key points

- Various methods of data labeling
  - Process feedback
  - Human labeling

- Advantages and disadvantages of both

Validating Data

Detecting Data Issues

Drift and skew

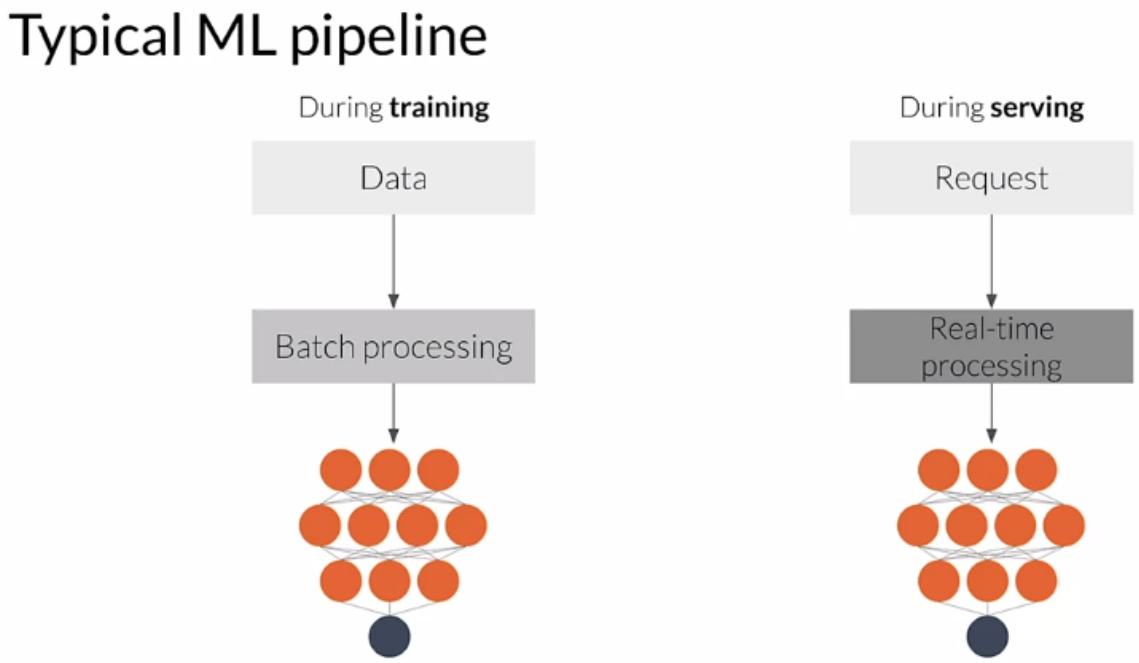
Drift

Changes in data over time, such as data collected once a day

Skew

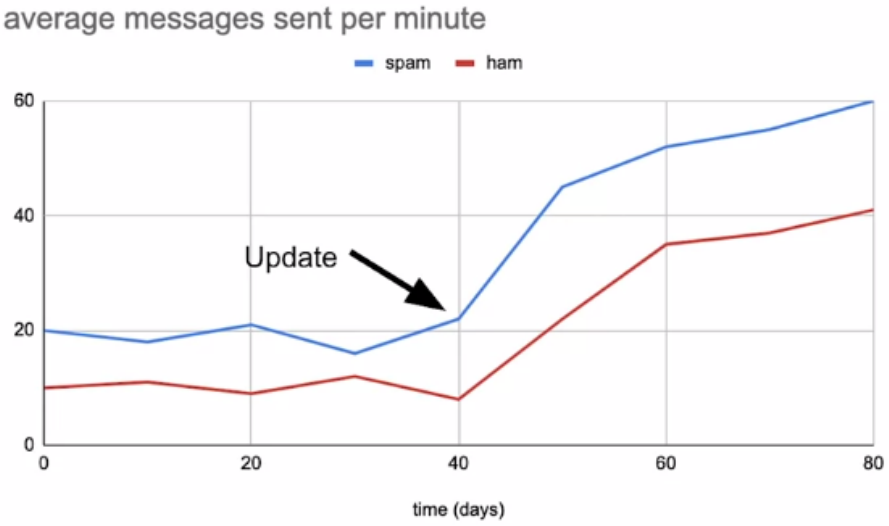
Difference between two static versions, or different sources, such as training set and serving set

For example, it could be the difference between your training set and the data that you're getting for prediction requests, your serving set. Those differences are referred to as skew. In a typical ML pipeline, this shows batch processing, but it could also be online processing. You'll have different sources of data that are conceptually the same. They have the same feature vector, but over time they will change. That means that model performance can either drop quickly due to things like system failure or can decay over time due to changes in the data and things like changes in the world. We're going to focus on performance decay over time that arises due to issues between training and serving data. There's really two main reasons for that. There's **Data drift**, which are changes in the data between training and serving typically and Concept drift, which are changes in the world changes in the ground truth.



To understand model decay over time, an ML model will start to perform poorly in many cases and we refer to this as model decay. That's usually or often caused by drift, which is, changes in a conceptual way. There is, changes in the statistical properties of the features. It's sometimes due to things like seasonality or trend or unexpected events, or just changes in the world. This example, here, we're looking at an app that during trainin,g the app classified as a spammer, any user who is sending 20 or more messages per minute. We classified anybody like that as a spammer. But after a system update which you see as labeled on the chart there, both spammers and non-spammers start to send more messages. In this case, the data, the world has changed and that causes unwanted misclassification. We have all of our users classified as spammers which they probably won't like.

Model Decay : Data drift

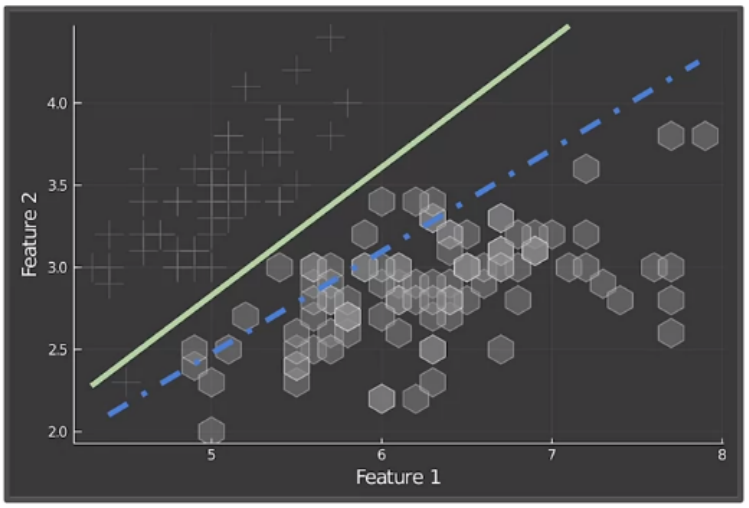


Concept drift is a change in the statistical properties of the labels over time. At training, an ML model learns a mapping between the features and the labels. In a static world that's fine, that won't change. But in the real-world, distribution and the labels meaning will change. The model needs to change as well as the mapping found during training will no longer be valid.

Detecting data issues

- Detecting schema skew
  - Training and serving data do not conform to the same schema
- Detecting distribution skew
  - Dataset shift → covariate or concept shift
- Requires continuous evaluation

Performance decay : Concept drift



Let's take a look at a more rigorous definition of the drift and skew that we're talking about. **Dataset shift** occurs when the joint probability of x are features and y are labels is not the same during training and serving. The data has shifted over time. **Covariate shift** refers to the change in distribution of the input variables present in training and serving data. In other words, it's where the marginal distribution of x are features is not the same during training and serving, but the conditional distribution remains unchanged. **Concept shift** refers to a change in the relationship between the input and output variables as opposed to the differences in the Data Distribution or input itself. In other words, it's when the conditional distribution of y are labels given x are features is not the same during training and serving, but the marginal distribution of x are features remains unchanged.

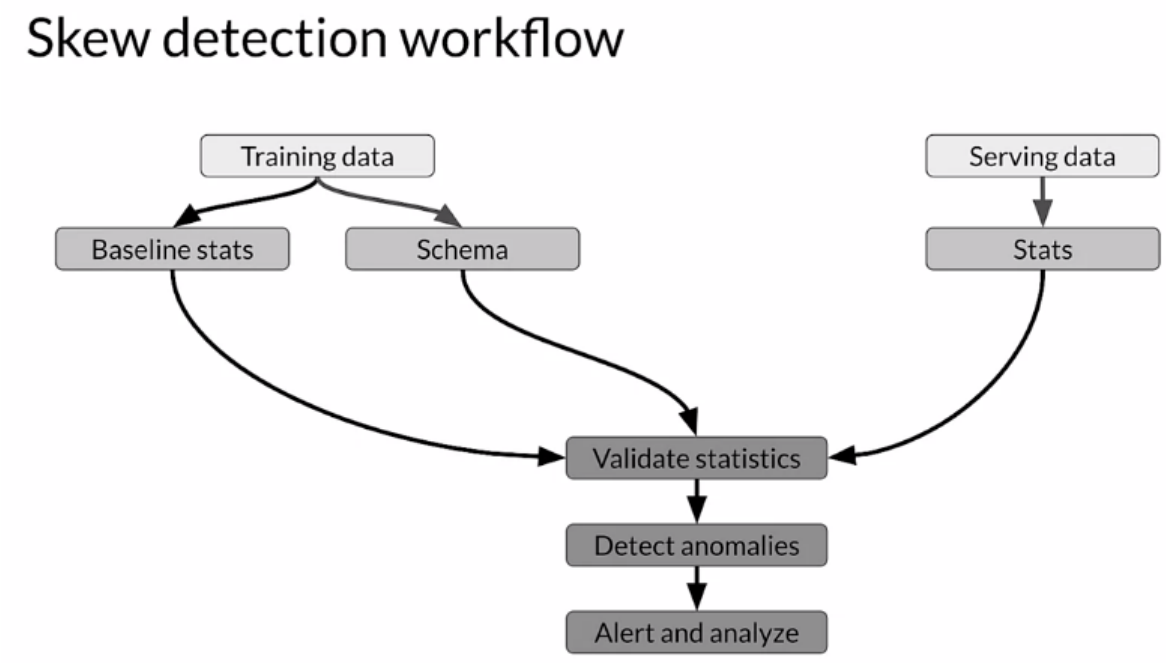
Detecting distribution skew

	Training	Serving
Joint	$P_{\text{train}}(y, x)$	$P_{\text{serve}}(y, x)$
Conditional	$P_{\text{train}}(y\|x)$	$P_{\text{serve}}(y\|x)$
Marginal	$P_{\text{train}}(x)$	$P_{\text{serve}}(x)$

Dataset shift $P_{\text{train}}(y, x) \neq P_{\text{serve}}(y, x)$

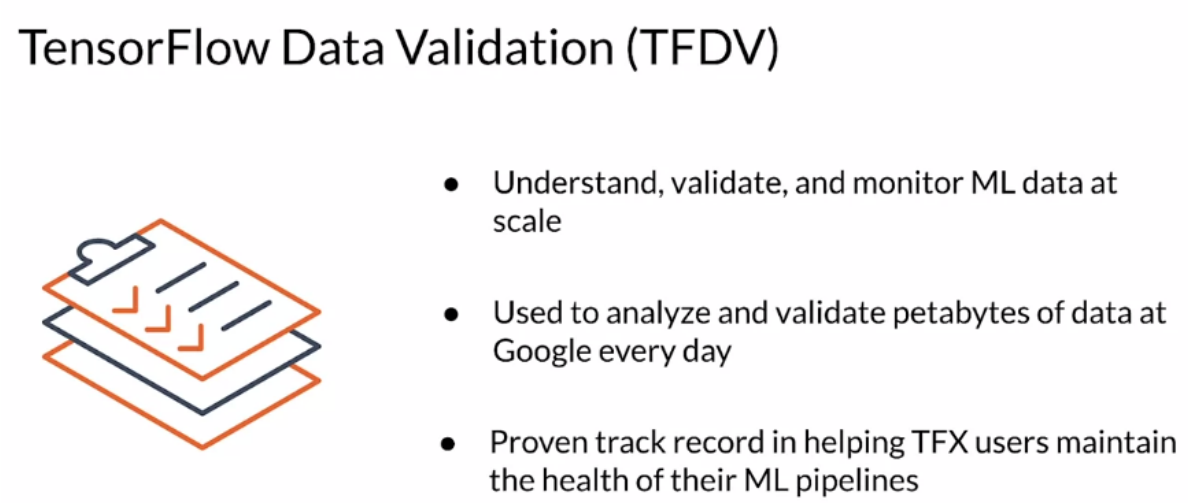
Covariate shift $P_{\text{train}}(y|x) = P_{\text{serve}}(y|x)$, $P_{\text{train}}(x) \neq P_{\text{serve}}(x)$

Concept shift $P_{\text{train}}(y|x) \neq P_{\text{serve}}(y|x)$, $P_{\text{train}}(x) = P_{\text{serve}}(x)$



There's a straightforward workflow to detect data skew. The first stage is looking at training data and computing baseline statistics and a reference schema. Then you do basically the same with your serving data, you're going to generate the descriptive statistics. Then you compare the two. You compare your serving baseline statistics and instances. You check for differences between that and your training data. You look for skew and drift. Significant changes become anomalies and they'll trigger an alert. That alert goes to whoever's monitoring system, that can either be a human or another system to analyze the change and decide on the proper course of action. That's got to be the remediation of the way that you're going to fix and react to that problem.

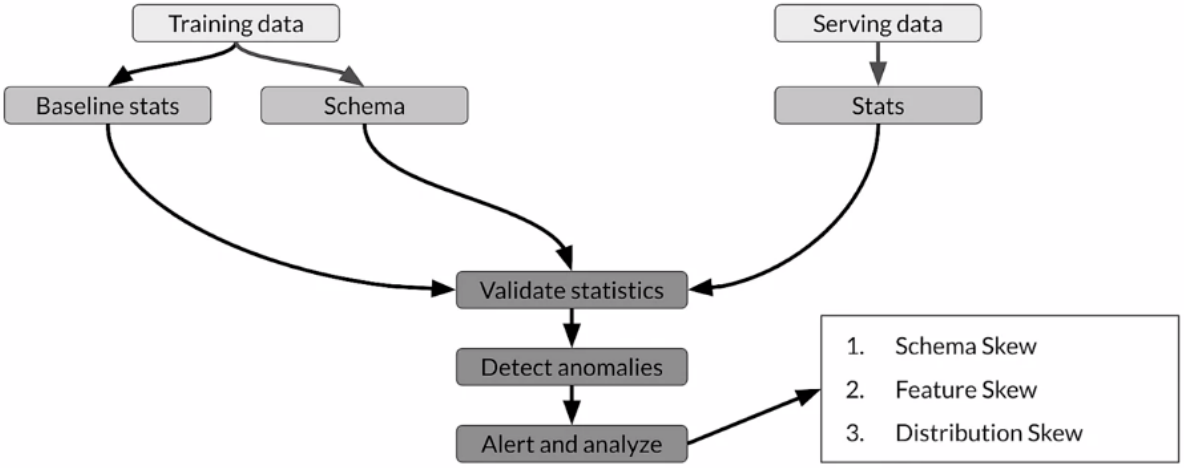
TensorFlow Data Validation



TFDV capabilities

- Generates data statistics and browser visualizations
- Infers the data schema
- Performs validity checks against schema
- Detects training/serving skew

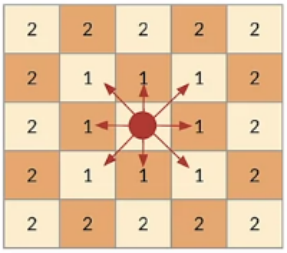
Skew detection - TFDV



Skew - TFDV

- Supported for categorical features
- Expressed in terms of L-infinity distance (Chebyshev Distance):

$$D_{\text{Chebyshev}}(x, y) = \max_i(|x_i - y_i|)$$



- Set a threshold to receive warnings

Schema skew

Serving and training data don't conform to same schema:

- For example, `int != float`

Feature skew

Training **feature values** are different than the serving **feature values:**

- Feature values are modified between training and serving time
- Transformation applied only in one of the two instances

Distribution skew

Distribution of serving and training dataset is significantly different:

- Faulty sampling method during training
- Different data sources for training and serving data
- Trend, seasonality, changes in data over time

Key points

- TFDV: Descriptive statistics at scale with the embedded facets visualizations
- It provides insight into:
  - What are the underlying statistics of your data
  - How does your training, evaluation, and serving dataset statistics compare
  - How can you detect and fix data anomalies

Wrap up

- Differences between ML modeling and a production ML system
- Responsible data collection for building a fair production ML system
- Process feedback and human labeling
- Detecting data issues

Practice data validation with TFDV in this week's exercise notebook

Test your skills with the programming assignment

Week 1 Optional References

Week 1: Collecting, Labeling and Validating Data

This is a compilation of optional resources including URLs and papers appearing in lecture videos. If you wish to dive more deeply into the topics covered this week, feel free to check out these optional references. You won't have to read these to complete this week's practice quizzes.

MLops

Data 1st class citizen

Runners app

Rules of ML

Bias in datasets

Logstash

Fluentd

Google Cloud Logging

AWS ElasticSearch

Azure Monitor

TFDV

Chebyshev distance

Papers

Konstantinos, Katsiapis, Karmarkar, A., Altay, A., Zaks, A., Polyzotis, N., … Li, Z. (2020). Towards ML Engineering: A brief history of TensorFlow Extended (TFX). http://arxiv.org/abs/2010.02013

Paleyes, A., Urma, R.-G., & Lawrence, N. D. (2020). Challenges in deploying machine learning: A survey of case studies. http://arxiv.org/abs/2011.09926

ML code fraction:

Sculley, D., Holt, G., Golovin, D., Davydov, E., & Phillips, T. (n.d.). Hidden technical debt in machine learning systems. Retrieved April 28, 2021, from Nips.cc https://papers.nips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf

Week 2: Feature Engineering, Transformation and Selection

Feature Engineering

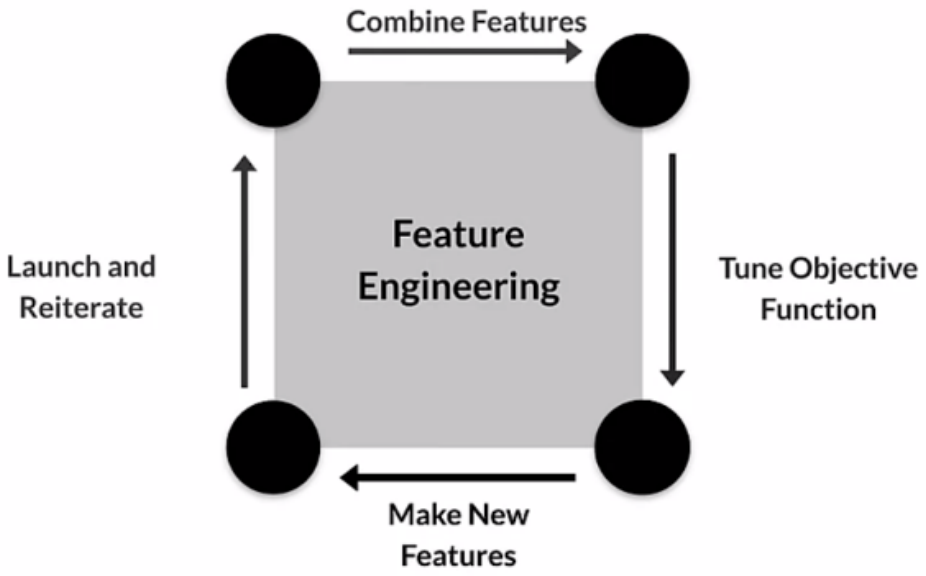
Introduction to Preprocessing

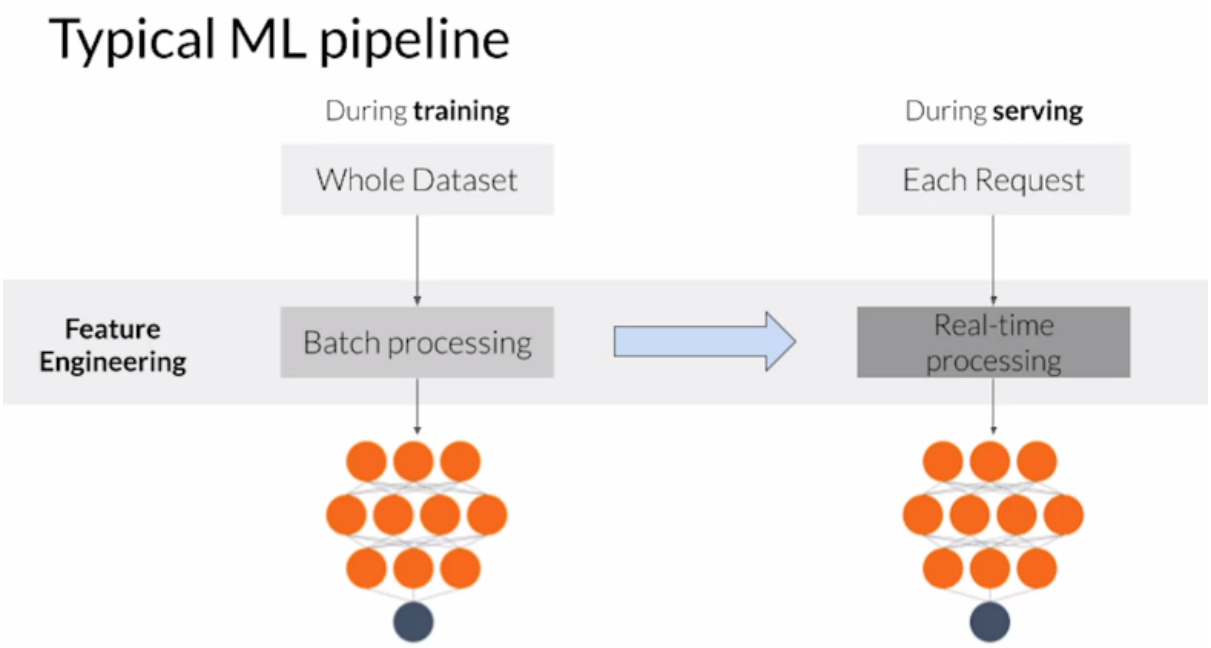
Squeezing the most out of data

- Making data useful before training a model
- Representing data in forms that help models learn
- Increasing predictive quality
- Reducing dimensionality with feature engineering

The art of feature engineering tries to improve your model's ability to learn while reducing, if possible, the compute resources it requires. It does this by transforming and projecting, eliminating and or combining the features in your raw data to form a new version of your data set. So typically across the ML pipeline, you incorporate the original features often transformed or projected to a new space and or combinations of your features.

Art of feature engineering

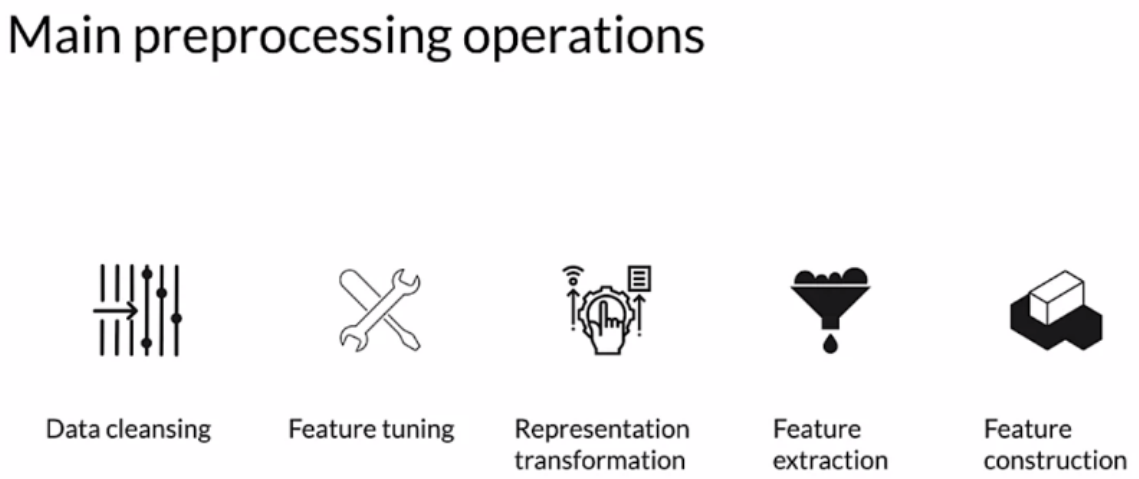




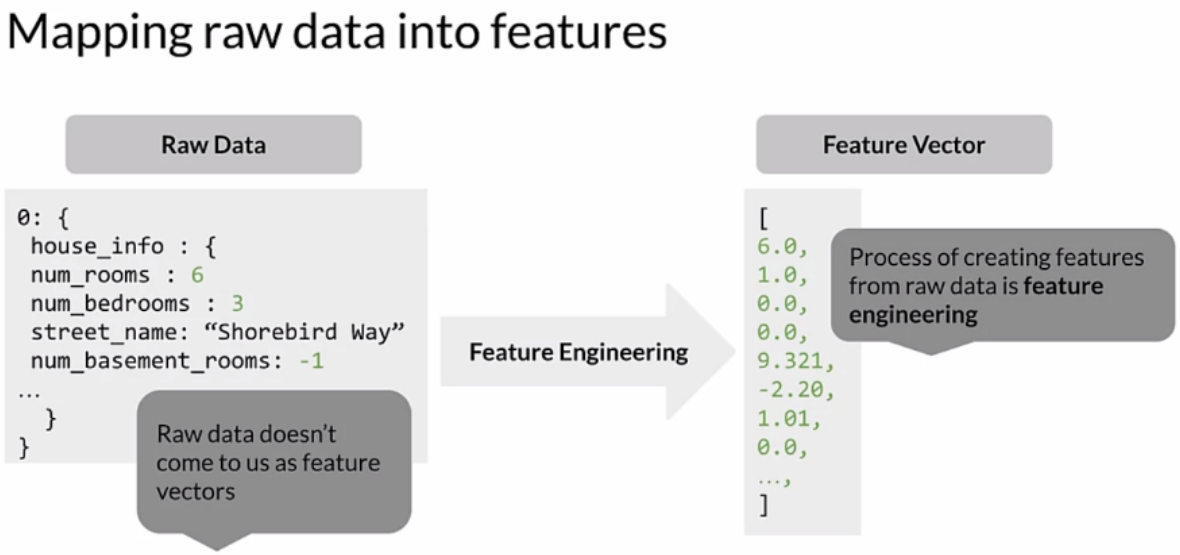
Feature engineering is usually applied in two fairly different ways, during training, you usually have the entire data set available to you. So you can use global properties of individual features in your feature engineering transformations. For example, you can compute the standard deviation of a feature and then use that to perform normalization. However, when you serve your trained model, you must do the same feature engineering so that you give your model the same types of data that it was trained on. For example, if you created a one hot vector for a categorical feature when you trained, you need to also create an equivalent one hot vector when you serve your model. However, during training and serving, you typically process each request individually, so it's important that you include global properties of your features, such as the standard deviation. If you use it during training include that with the feature engineering that you do when serving, failing to do that right is a very common source of problems in production systems, and often these errors are difficult to find.

Key points

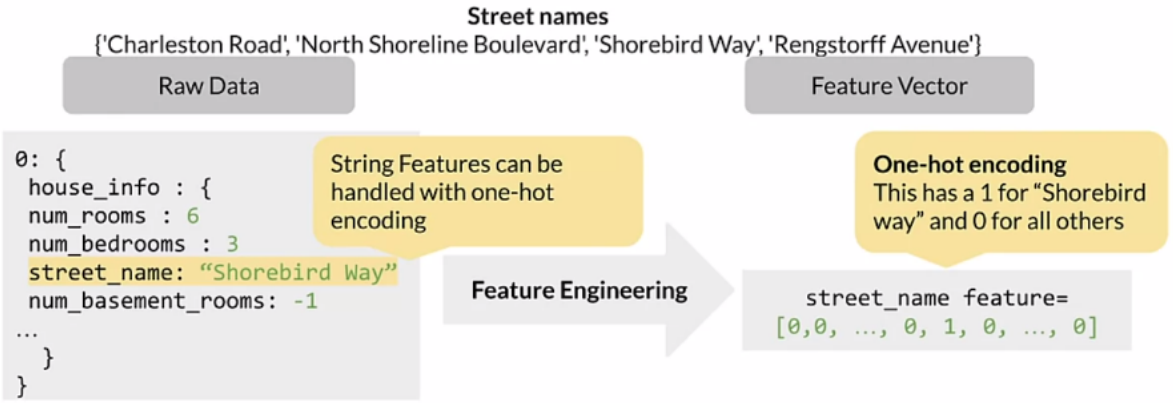
- Feature engineering can be difficult and time consuming, but also very important to success
- Squeezing the most out of data through feature engineering enables models to learn better
- Concentrating predictive information in fewer features enables more efficient use of compute resources
- Feature engineering during training must also be applied correctly during serving



One of the most important Preprocessing Operations is **Data cleansing**, which in broad terms consists in eliminating or correcting erroneous data. You'll often need to perform **Transformations** on your data, so scaling or normalizing your numeric values, for example. Since models, especially neural networks, are sensitive to the amplitude or range of numerical features, data preprocessing helps Machine Learning build better predictive models. **Dimensionality Reduction** involves reducing the number of features by creating lower dimension and more robust data representations. **Feature Construction** can be used to create new features by using several different techniques, which we'll talk about some of them.



Mapping categorical values



Tensorflow provides three different functions for creating columns of categorical vocabulary and other frameworks do very similar things. A categorical column with vocabulary lists maps each string to an integer based on an explicit vocabulary list. Feature name is a string which corresponds to the categorical feature and vocabulary list is an ordered list that defines vocabulary. Feature or categorical column with a vocabulary file very similar from a file is used when you have two long lists and this function allows you to put the words in a separate file. In this case, vocabulary list is defined as a file that will define the list of words. This is very typical for working with vocabularies.

Categorical Vocabulary

```
# From a vocabulary list

vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_list(

                            key=feature_name,

                            vocabulary_list=["kitchenware", "electronics", "sports"])
# From a vocabulary file

vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_file(

                             key=feature_name,

                             vocabulary_file="product_class.txt",

                             vocabulary_size=3)
```

Empirical knowledge of data

Text - stemming, lemmatization, TF-IDF, n-grams, embedding lookup

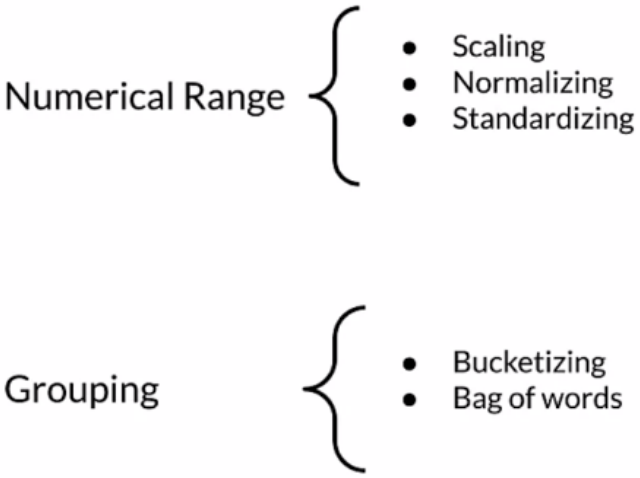
Images - clipping, resizing, cropping, blur, Canny filters, Sobel filters, photometric distortions

Key points

- Data preprocessing: transforms raw data into a clean and training-ready dataset
- Feature engineering maps:
  - Raw data into feature vectors
  - Integer values to floating-point values
  - Normalizes numerical values
  - Strings and categorical values to vectors of numeric values
  - Data from one space into a different space

Feature Engineering Techniques

Feature engineering techniques



Scaling

- Converts values from their natural range into a prescribed range
  - E.g. Grayscale image pixel intensity scale is [0,255] usually rescaled to [-1,1]

```
image = (image - 127.5) / 127.5
```

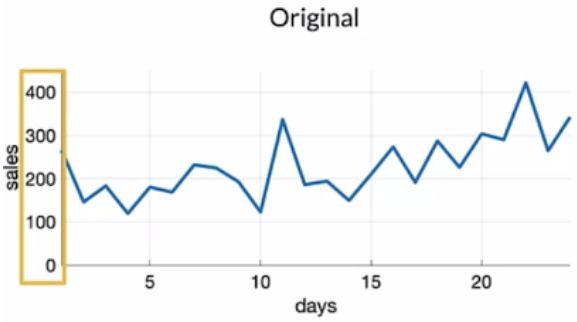
- Benefits
  - Helps neural nets converge faster
  - Do away with NaN errors during training
  - For each feature, the model learns the right weights

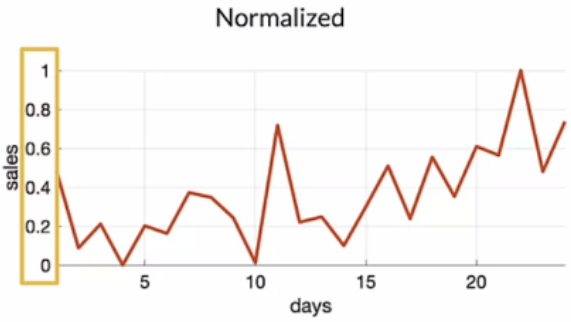
Normalizations are usually good if you know that the distribution of your data is not Gaussian. Doesn't always have to be true, but typically that's a good assumption to start with. A good rule of thumb. If you're working with data that you know is not Gaussian, or a normal distribution, then normalization is a good technique to start with. Normalization is widely used for scaling, you have a numerical feature that might start out like this, where numbers are falling and there is a kind of a range. You're going to transform that to the normalized version of that, which is bounded in a range between 0-1.

Normalization

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

$$X_{\text{norm}} \in [0, 1]$$



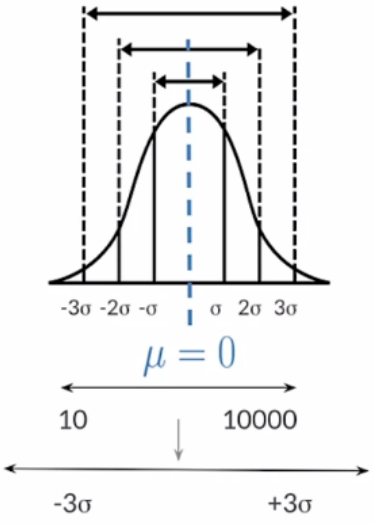


Standardization (z-score)

- Z-score relates the number of standard deviations away from the mean
- Example:

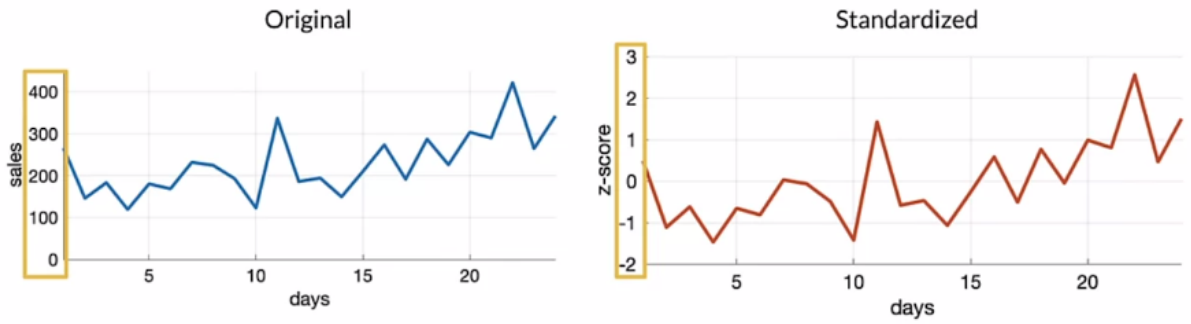
$$X_{\text{std}} = \frac{X - \mu}{\sigma} \quad \text{(z-score)}$$

$$X_{\text{std}} \sim \mathcal{N}(0, \sigma)$$

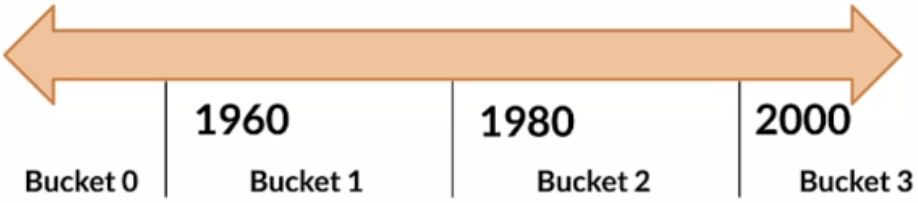


Standardization, which is often using a Z-score, is a different technique. It's a way of scaling using the standard deviation. It's looking at the distribution itself and trying to scale relative to that. The example here is using, , and you're going to subtract the mean and divide by the standard deviation. That gives you the Z- score or the standardized value of X. Which is somewhere between zero and the standard deviation. This is how that's expressed, actually

between some multiple of the standard deviation. But it's centered around the mean of the data. If the original looked like this, again, a standardized value of that might look like this. Notice that this score is centered on zero, so the mean is translated to zero. But you can have negative values and positive values that are beyond one. It's a little bit less bounded transformation than a normalization is. But there are some advantages to it, that your data is a normal distribution, then a standardization technique is a good place to start, it's a good rule of thumb to start with for your numerical features. But I encourage you to try both standardization and normalization and compare the results. Because sometimes it doesn't make much of a difference. Sometimes it can make a substantial difference. It's good to try both.

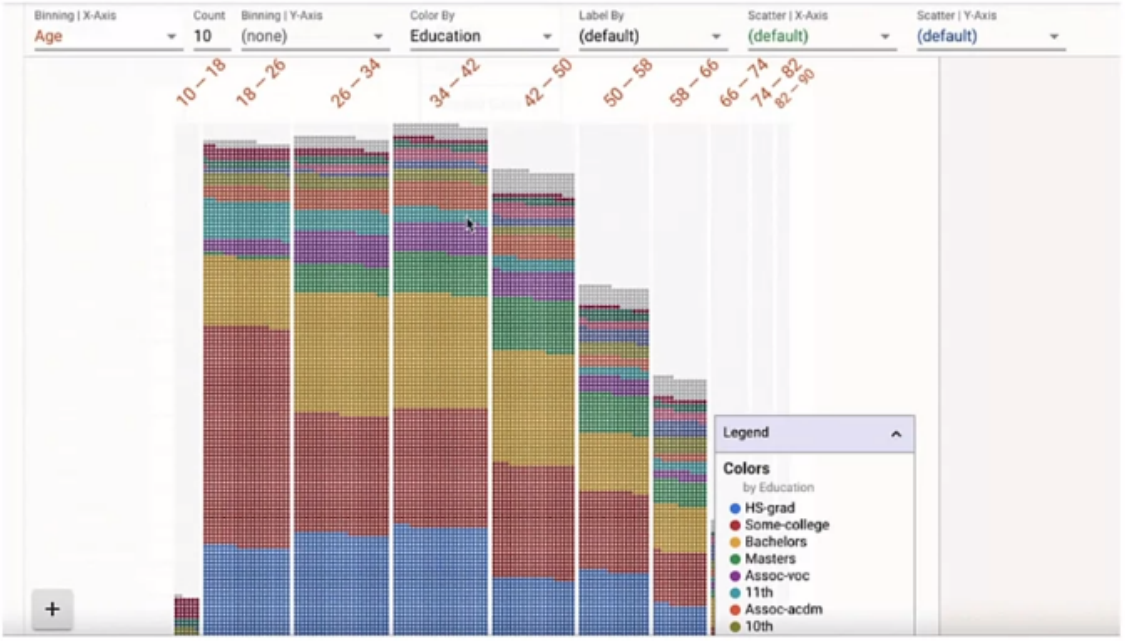


Bucketizing / Binning



Date Range	Represented as...
< 1960	[1, 0, 0, 0]
>= 1960 but < 1980	[0, 1, 0, 0]
>= 1980 but < 2000	[0, 0, 1, 0]
>= 2000	[0, 0, 0, 1]

Binning with Facets



Other techniques

Dimensionality reduction in embeddings

- Principal component analysis (PCA)
- t-Distributed stochastic neighbor embedding (t-SNE)
- Uniform manifold approximation and projection (UMAP)

Feature crossing

TensorFlow embedding projector

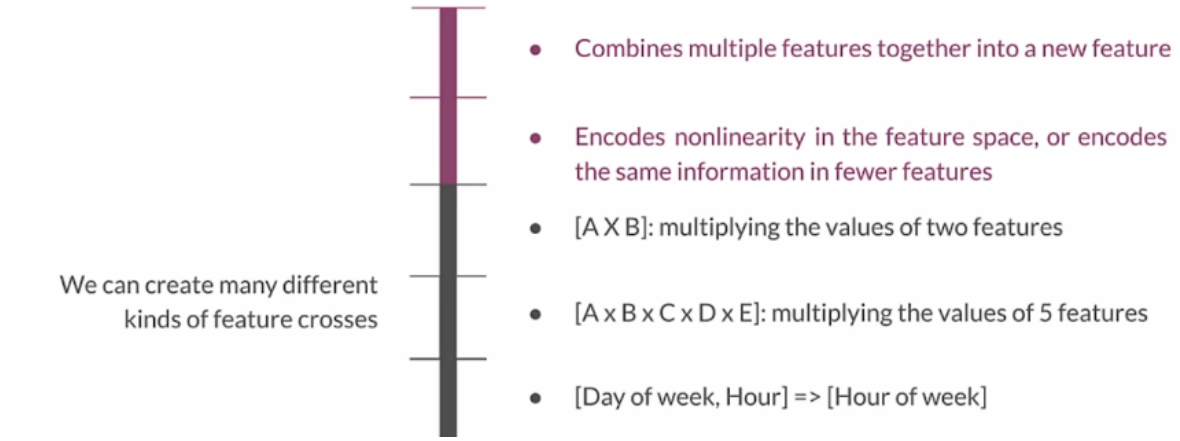
- Intuitive exploration of high-dimensional data
- Visualize & analyze
- Techniques
  - PCA
  - t-SNE
  - UMAP
  - Custom linear projections
- Ready to play

Key points

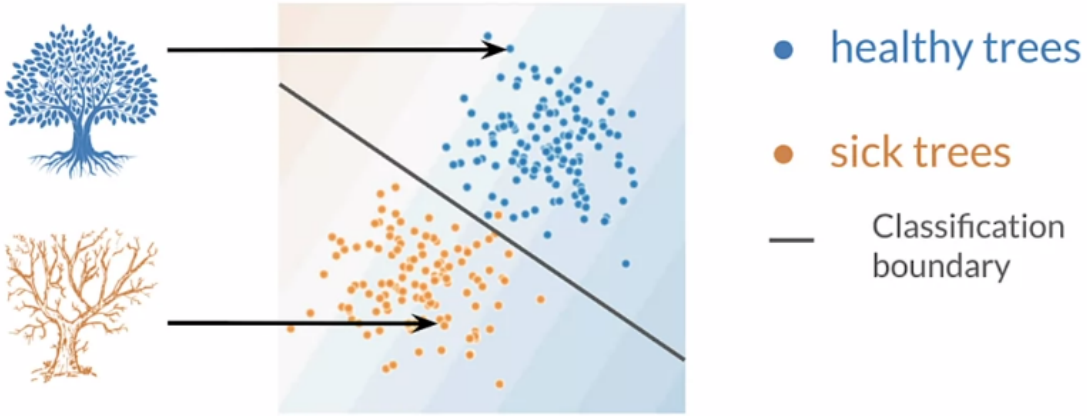
- Feature engineering:
  - Prepares, tunes, transforms, extracts and constructs features.
- Feature engineering is key for model refinement
- Feature engineering helps with ML analysis

Feature Crosses

Feature crosses

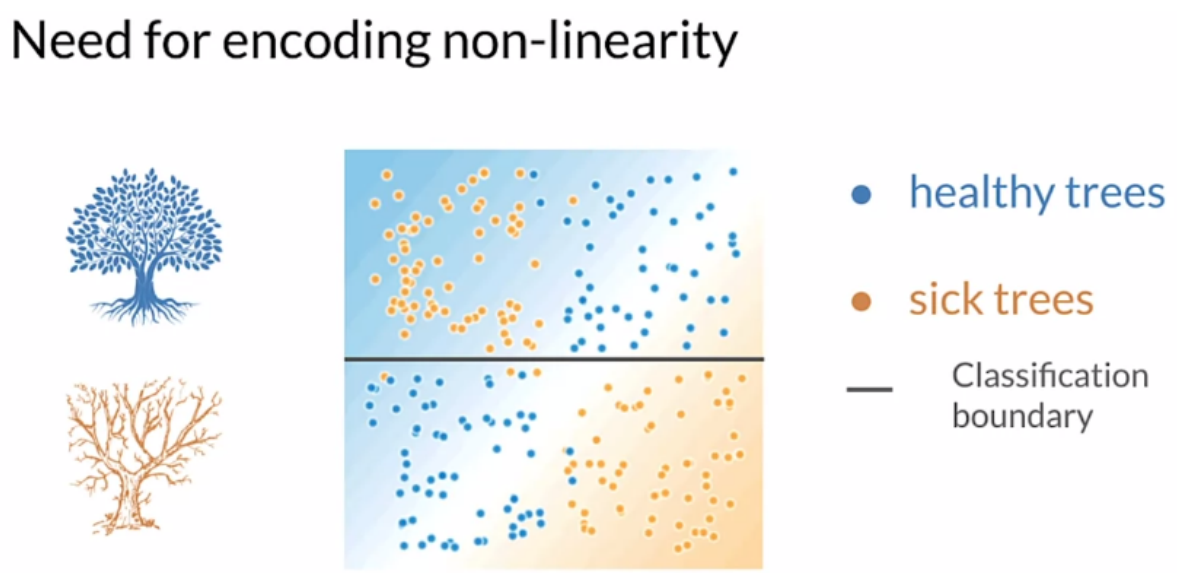


Encoding features



We're going to use a scatter-plot to try to understand our data. We're looking at the scatter plot and we use some visualization tools, and we ask ourselves, can we draw a line to separate these two groups, these two clusters? If we can use a line to create the decision boundary, then we know that we can use a linear model. That's great. In this case, looking at this, we could use a line to do that separation. That's great. Linear models are very efficient. We love that, but let's suppose the data looks like this. Not so easy anymore. This becomes a non-linear program. Then the question is, can we project this data into a linear space and use it with a linear model, or do we have to use a non-linear model to work with this data? Because if we try to draw just a linear classification boundary with the data as is, it doesn't really work too great. Again, we can use visualization tools to help us understand this. Looking at our data really helps inform us and guide the choices that we make as developers about what kind of models we choose and what feature engineering we apply. Key points here, Feature Crossing as a way to create synthetic features, often encoding non-linearity in the features space. We're going to transform both

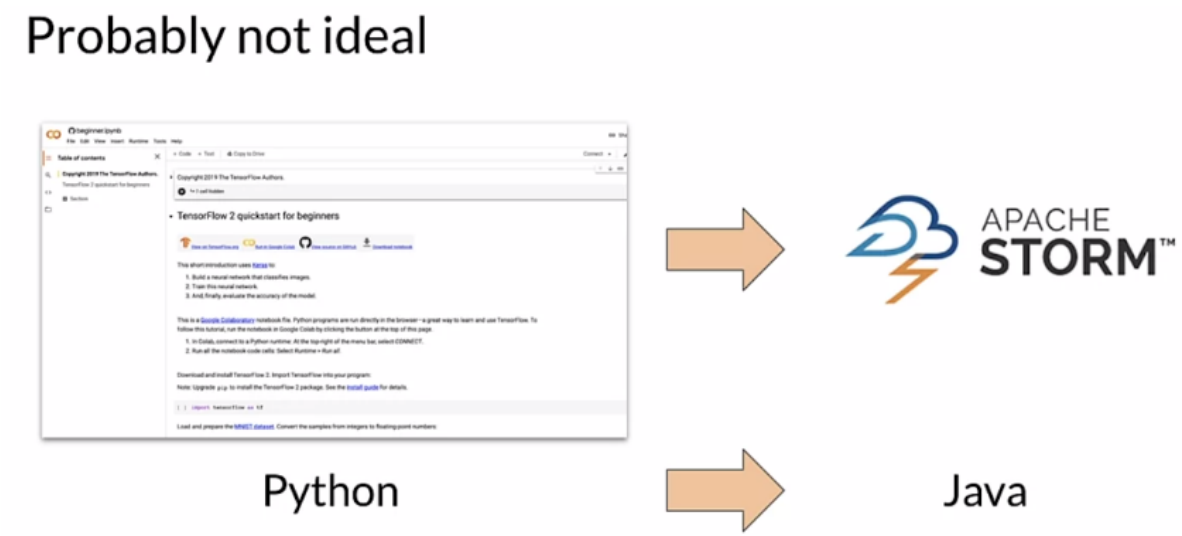
categorical and numerical. We could do that in, into either continuous variables or the other way around.



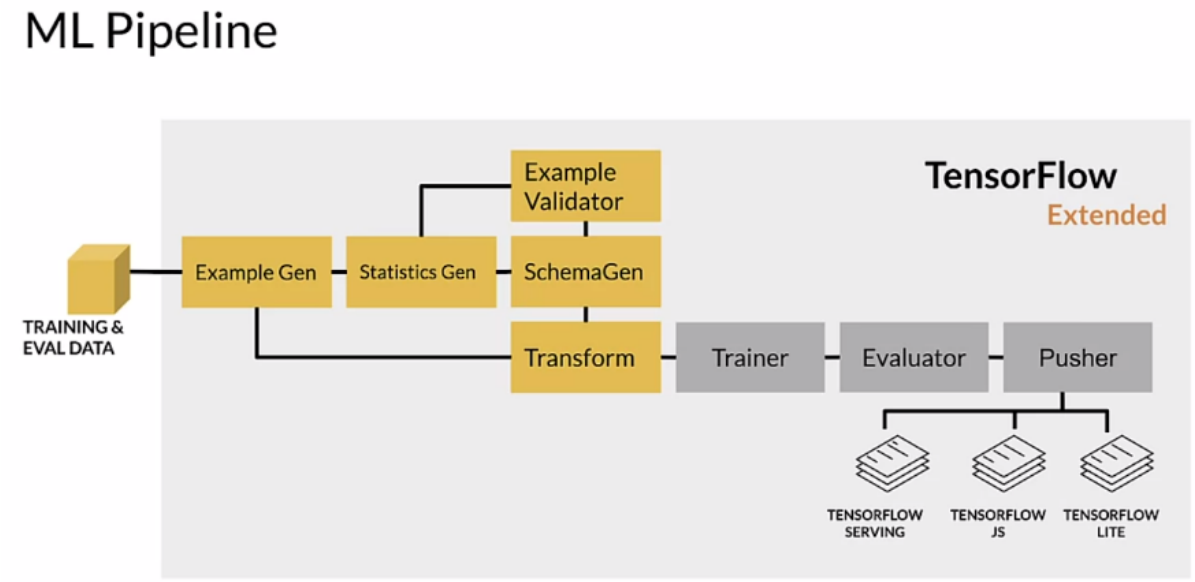
Feature Transformation at Scale

Preprocessing Data at Scale

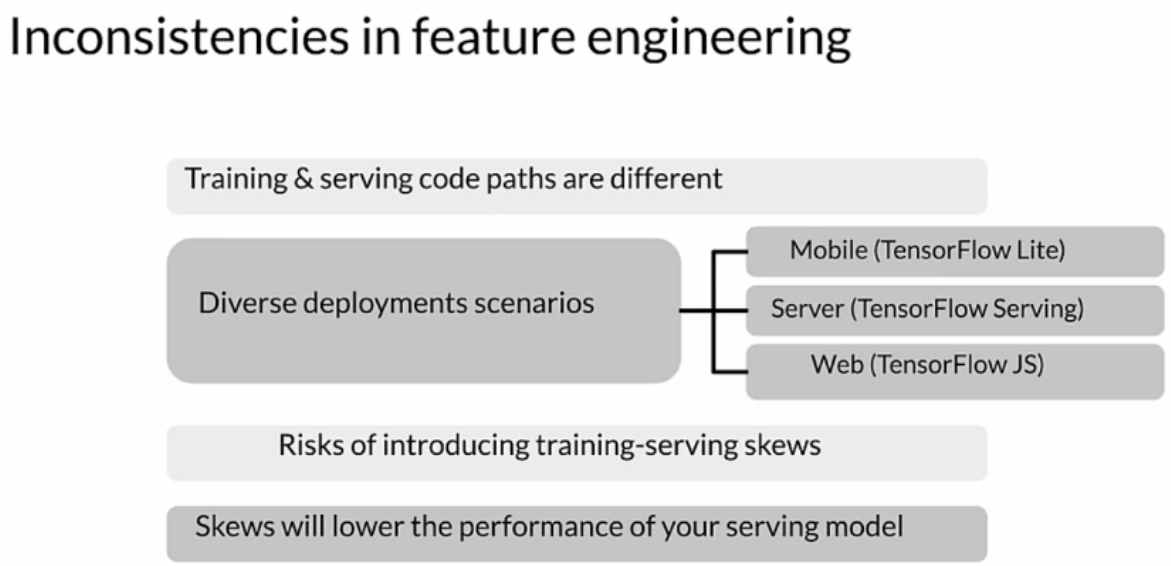
We were developing our notebooks in python and then when we deployed into storm we had to translate all of the feature engineering that we did into java. Well, as you can imagine, that was not ideal and there were little weird problems that cropped up often very difficult to find doing that transformation. This is not an ideal way to do production machine learning.



So what is a much better technique is to use a pipeline, a unified framework where you can both train and deploy with consistent and reproducible results.



But there's a number of challenges that you need to deal with. So to preprocessed data at scale, we start with real world models and these could be terabytes of data. So when you're doing this kind of kind of work, you want to start with a subset of your data work out. As many issues as you can think about how that's going to scale up to the terabytes that you need to actually work with your whole dataset. Large scale data processing frameworks are no different than what you're going to use, on your laptop or in a notebook or what have you. So you need to start thinking about that from the beginning of the process, how this is going to be applied and the earlier you can start working with those frameworks. The more you work out the issues early with smaller datasets and quicker turnaround time, consistent transformations between training and serving are incredibly important. If you do different transformations when you're serving your model than you did when you were training your model, you are going to have problems and often those problems are very hard to find.



So there is a notion of the granularity at which you're doing preprocessing. So you need to think about this, you're going to do transformations, both the instance level and a full pass over your data. And depending on the transformation that you're doing, you may be doing it in one or the other. You can usually always do instance level. But a full path requires that you have the whole dataset. So for clipping, even for clipping, you need to set clipping boundaries. If you're not doing that through some arbitrary setting of those boundaries, if you're using the data set itself to determine how to clip, then you're going to need to do a full pass. So min max for clipping is going to be important. Doing a multiplication at the instance level is fine, but scaling well now I'm going to need probably the standard deviation and maybe the min and max. Bucketizing similarly, unless I know ahead of time what buckets are going to be, I'm going to need to do a full pass to find what buckets make sense. Expanding features I can probably do that at the instance level. So these two things are different at training time. I have the whole dataset so I can do a full pass, although it can be expensive to do that. At serving time, I get individual examples, so I can only really do instance level. So anything I need to do that requires characteristics of the whole dataset. I need to have that saved off so I can use it at serving time.

Preprocessing granularity

Transformations	
Instance-level	**Full-pass**
Clipping	Minimax
Multiplying	Standard scaling
Expanding features	Bucketizing
etc.	etc.

When do you transform?

Pre-processing training dataset

Pros	Cons
Run-once	Transformations reproduced at serving
Compute on entire dataset	Slower iterations

So when do you transform, you're going to pre-process your training dataset and there's pros and cons in how you do that. First of all, you don't you only run **once per training session**, so that's cool. And you're going to compute it on the entire dataset, but only once, for each training session. The cons: well, you're going to have to reproduce all those transformations that serve or save off the information that you learned about the data, like the standard deviation. So that you can use that later while you're serving. And there's, slower iterations around this. Each time you make a change, you've got to make a full pass over the dataset.

So you can do this **within the model**. Transforming within the model has some nice features so there are cons as well. First of all it makes iteration somewhat easier because it's embedded as part of your model and there's guarantees around the transformation that you're doing. But the cons are it can be expensive to do those transforms, especially when your compute resources are limited. And think about when you do a transform within the model, you're going to do that same transform at serving time. So you may have say GPUs or TPUs when you're training, you may not when you're serving. So there's long model latency, that's when you're serving your model, if you're doing a lot of transformation with it that can slow down the response time for your model and increase latency.And, you can have transformations per batch that skew that we talked about. If you haven't saved those constants that you need, that could be an issue.

How about ‘within’ a model?

Transforming within the model

Pros	Cons
Easy iterations	Expensive transforms
Transformation guarantees	Long model latency
	Transformations per batch: skew

You can also **transform per batch** instead of for the entire dataset. So you could for example, normalize features by their average within the batch. That only requires you to make a pass over each batch and not the full data set, which when you're working with terabytes of data. That can be a significant advantage in terms of processing. And there's ways to normalize per batch as well. So you can compute an average and use that and normalization per batch and then do it again for the next batch there will be differences batch to batch. Sometimes that's good in cases. So for example where you have changes over time in a time series, that can be a good thing but you need to consider that. So normalizing by the average per batch, precomputing the average and using it during normalization, you can use it for multiple batches for example. So this is a different way to think about how to work with your data when you have a large dataset.

Why transform per batch?

- For example, normalizing features by their average
- Access to a single batch of data, not the full dataset
- Ways to normalize per batch
  - Normalize by average within a batch
  - Precompute average and reuse it during normalization

Optimizing instance-level transformations

- Indirectly affect training efficiency
- Typically accelerators sit idle while the CPUs transform
- Solution:
    - Prefetching transforms for better accelerator efficiency

Summarizing the challenges

- Balancing predictive performance
- Full-pass transformations on training data
- Optimizing instance-level transformations for better training efficiency (GPUs, TPUs, ...)

So to **summarize** the challenges we have to balance the predictive performance of our model and the requirements for it. Making full pass transformations on the training data is one thing. If we're going to do that then we need to think about how that works when we serve our model as well and save those constants. And we want to optimize the instance level transformations for better training efficiency. So things like prefetching can really help with that.

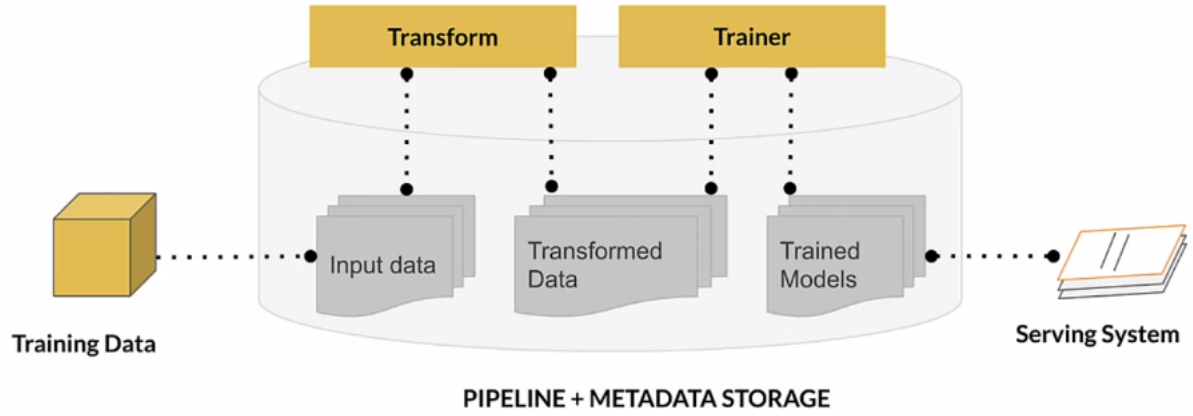
So **key points**, inconsistent data affects the accuracy of the results. So scalable data processing frameworks allow processing of large datasets and distributed inefficient ways. But we also need to think about how that gets applied in serving.

Key points

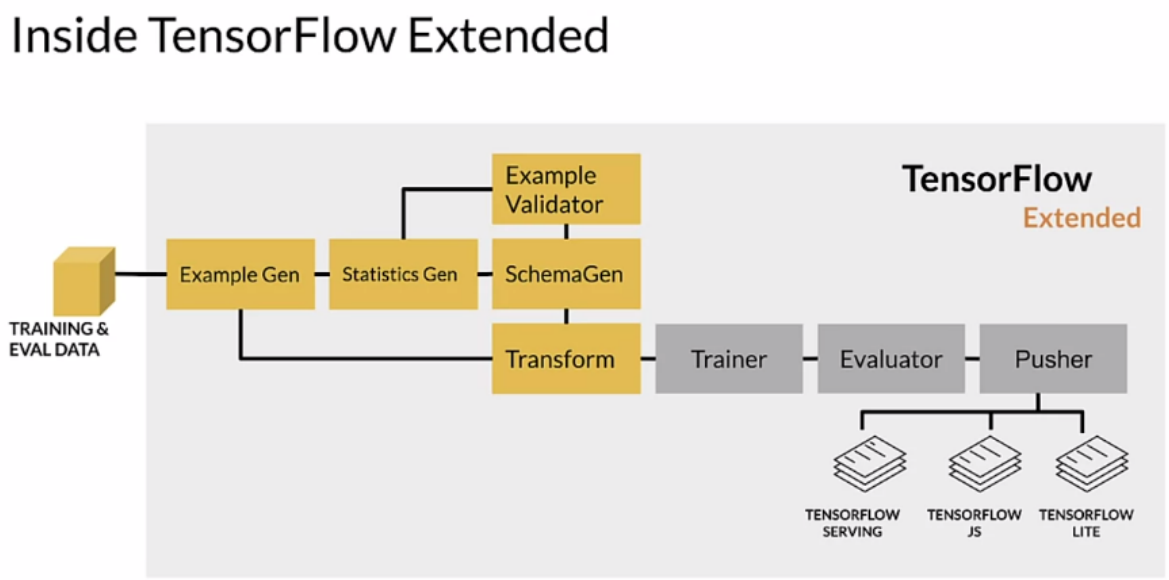
- Inconsistent data affects the accuracy of the results
- Need for scaled data processing frameworks to process large datasets in an efficient and distributed manner

TensorFlow Transform

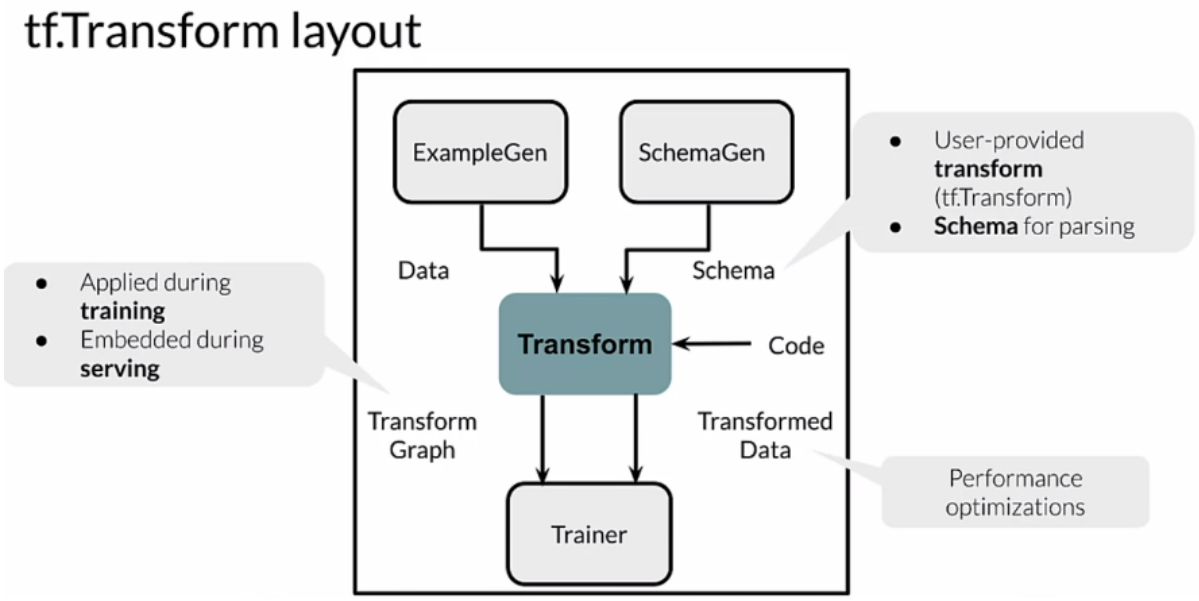
Enter tf.Transform



Looking at this a little differently, we're starting with our training and eval data. We've actually split our dataset. We split it with **ExampleGen**. Those get fed to **StatisticsGen**. These are both TFX components within a TFX pipeline. StatisticsGen calculates statistics for our data. For each of the features, if they're numeric features, for example, what is the mean of that feature value? What is the standard deviation? The min, the max, so forth. Those statistics get fed to **SchemaGen** which infers the types of each of our features. That creates a schema that is then used by downstream components including **Example Validator**, which takes those statistics and that schema, and it looks for problems in our data. We won't go into great detail here about the things that it looks for, but if we have examples that are the wrong type in a particular feature, so maybe we have an integer where we expected a float. Now, we're getting into **Transform**. Transform is going to take the schema that was generated on the original split dataset, and it's going to do our feature engineering. Transform is where the feature engineering happens. That gets given the **Trainer**, there's **Evaluator** that evaluates the results, a set of **Pusher** that pushes it to our deployment targets which is, however we're serving our models, so **TENSORFLOW SERVING**, or **JS**, or **LITE**, wherever it is that we're serving our model.

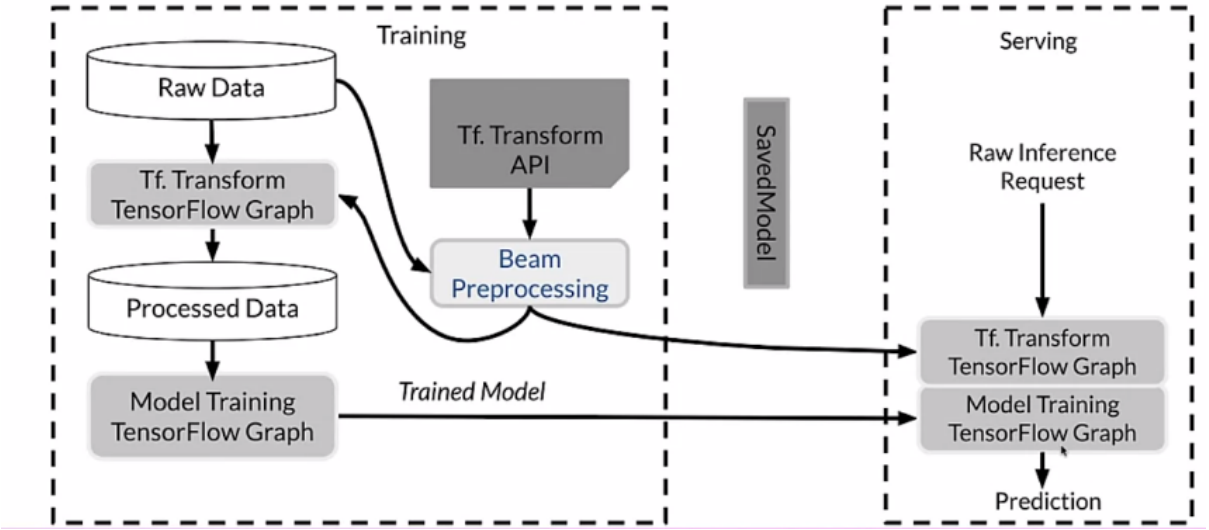


Internally, if we want to look at the transform component, it's getting inputs from, as we saw, ExampleGen and SchemaGen. That is the data that was originally split by Example Gen, and the schema that was generated by SchemaGen. That schema, by the way, may very well have been reviewed and improved by a developer who knew more about what to expect from the data than can be really inferred by SchemaGen. That's referred to as curating the schema. Transform gets that and it also gets a lot of user code because we need to express the feature engineering we want to do. If we're going to normalize a feature, we need to give it user code to tell transform to do that. The result is a TensorFlow graph, which was referred to as the transform graph and the transform data itself. The graph expresses all of the transformations that we are doing on our data, as a TensorFlow graph. The transform data is simply the result of doing all those transformations. Those are given to Trainer, which is going to use the transformed data for training and it's going to include the transform graph and we'll take a look at that in a second. We have a user provided transform component and we have a schema for it. We apply our transformations during training and as we'll see later, we also apply those transformations at serving time. There's a lot of performance optimizations that we apply as well, and we'll take a look at that in a second.

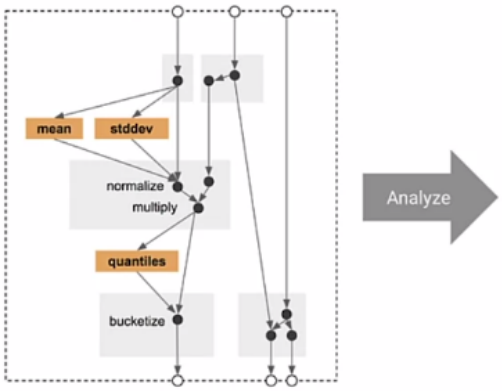


Continuing a little deeper here, we have our raw data and we have our transform component. It produces a TensorFlow graph. The result of running that graph is processed data, it's been transformed. That gets given in model training, as processed data is given to our trainer component, where we use it for training our model. Training our model creates a TensorFlow graph. Notice now we have two different graphs. We have a graph here from Transform and a graph here from Training. Using the Tf Transform API, we express the feature engineering that we want to do and we give that code, or rather the transform component gives that code to a Apache Beam distributed processing cluster. That way we can do, and remember this is all designed to work with potentially terabytes of data. That could be quite a bit of processing to do all of that transformation using a distributed processing cluster by using Apache Beam gives us the capacity to do that. The result is a saved model. Saved model is just a format that expresses a trained model as a saved model. That gets included, we have both the transform graph and the training graph. Those are given to our serving infrastructure. Now we have both of those graphs, the feature engineering that we did, and the model itself, the trained model and those are used when we serve requests.

tf. Transform: Going deeper



tf.Transform Analyzers



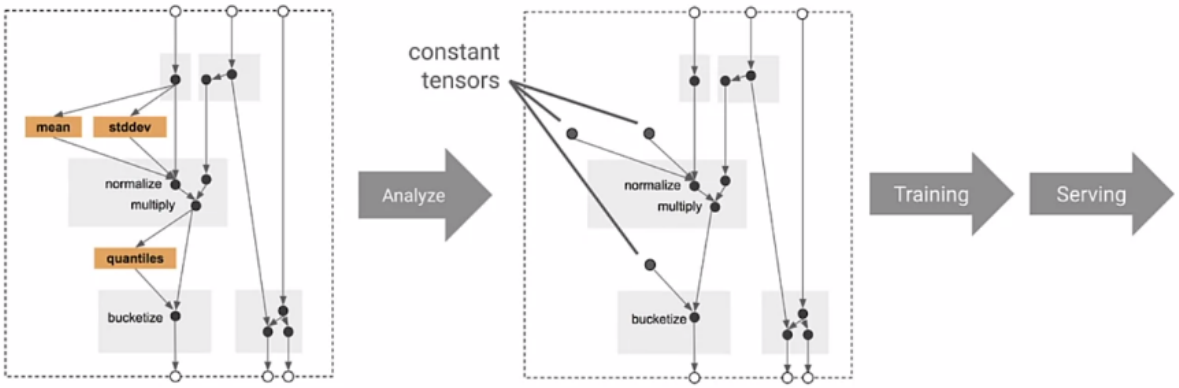
They behave like TensorFlow Ops, but run only once during training

For example:
tft.min *computes the minimum of a tensor over the training dataset*

Looking at this a little differently, we have analyzers. Again, we're using TensorFlow and we're using the TensorFlow Transform API. We're using TensorFlow Ops and a big part of what they do, what an Analyzer does is it makes a full pass over our dataset in order to collect constants that we're going to need when we do feature engineering. For example, if we're going to do a min-max, well, we need to make a full pass through our data to know what the min and the max are for each feature that we're using for that. Those are constants that we need to express. We're going to use an Analyzer to make that pass over the data and collect those constants. It's also going to express the operations that we're going to do. They behave like TensorFlow Ops, but they only were on once during training and then they're saved off as a graph. For example, if we're using the tft.min, which is one of the methods in the transform STK, it'll compute the minimum of a tensor over the training dataset.

Looking at how that gets applied, we have the graph that is our feature engineering. We run an analysis across our dataset to collect the constants that we need. Really what we're doing is we're enabling us later to apply these into our Transform graph, so that we can transform individual examples without making a pass over our entire dataset. That gets applied during training, and the same exact graph gets applied during serving. There is no potential here for training and serving skew or having different code paths when we train our model versus when we serve our model that causes problems and those not be equivalent. We're using exactly the same graph in both places, so there is no potential for doing that.

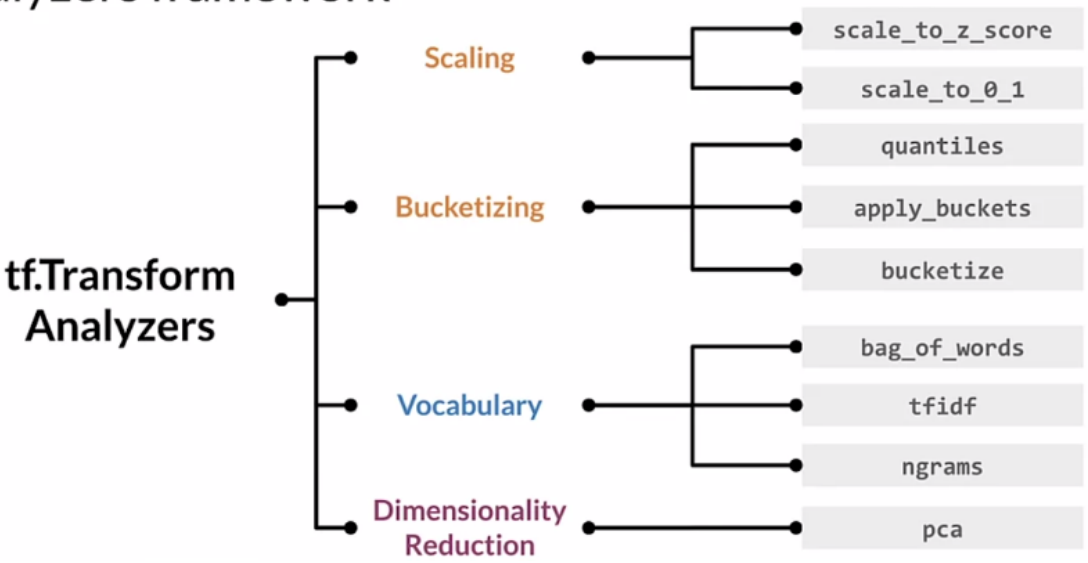
How Transform applies feature transformations



Benefits of using tf.Transform

- Emitted tf.Graph holds all necessary constants and transformations
- Focus on data preprocessing only at training time
- Works in-line during both training and serving
- No need for preprocessing code at serving time
- Consistently applied transformations irrespective of deployment platform

Analyzers framework



tf.Transform preprocessing_fn

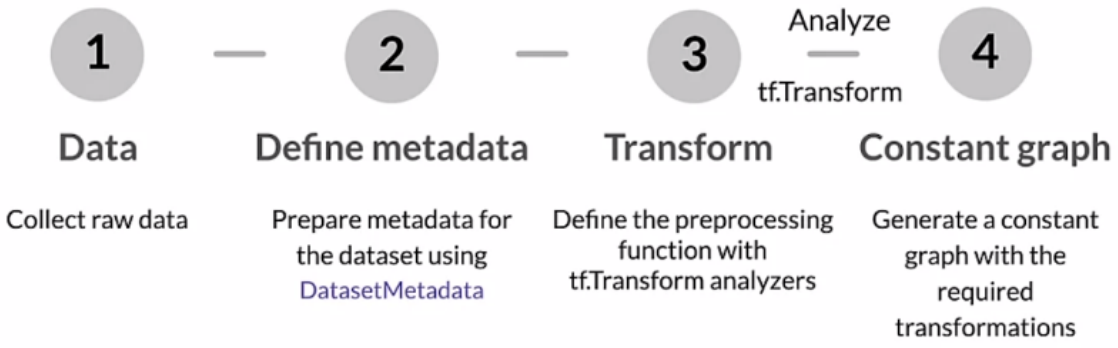
```
def preprocessing_fn(inputs):
    ...

for key in DENSE_FLOAT_FEATURE_KEYS:
    outputs[key] = tft.scale_to_z_score(inputs[key])

for key in VOCAB_FEATURE_KEYS:
    outputs[key] = tft.vocabulary(inputs[key], vocab_filename=key)

for key in BUCKET_FEATURE_KEYS:
   outputs[key] = tft.bucketize(inputs[key], FEATURE_BUCKET_COUNT)
```

Commonly used imports

```
import tensorflow as tf
import apache_beam as beam
import apache_beam.io.iobase

import tensorflow_transform as tft
import tensorflow_transform.beam as tft_beam
```

Hello World with tf.Transform

Hello world with tf.Transform



Collect raw samples (Data)

```
[
   {'x': 1, 'y': 1, 's': 'hello'},
   {'x': 2, 'y': 2, 's': 'world'},
   {'x': 3, 'y': 3, 's': 'hello'}
]
```

Inspect data and prepare metadata (Data)

```
from tensorflow_transform.tf_metadata import (
    dataset_metadata, dataset_schema)

raw_data_metadata = dataset_metadata.DatasetMetadata(
    dataset_schema.from_feature_spec({

    'y': tf.io.FixedLenFeature([], tf.float32),
        'x': tf.io.FixedLenFeature([], tf.float32),
        's': tf.io.FixedLenFeature([], tf.string)
    }))
```

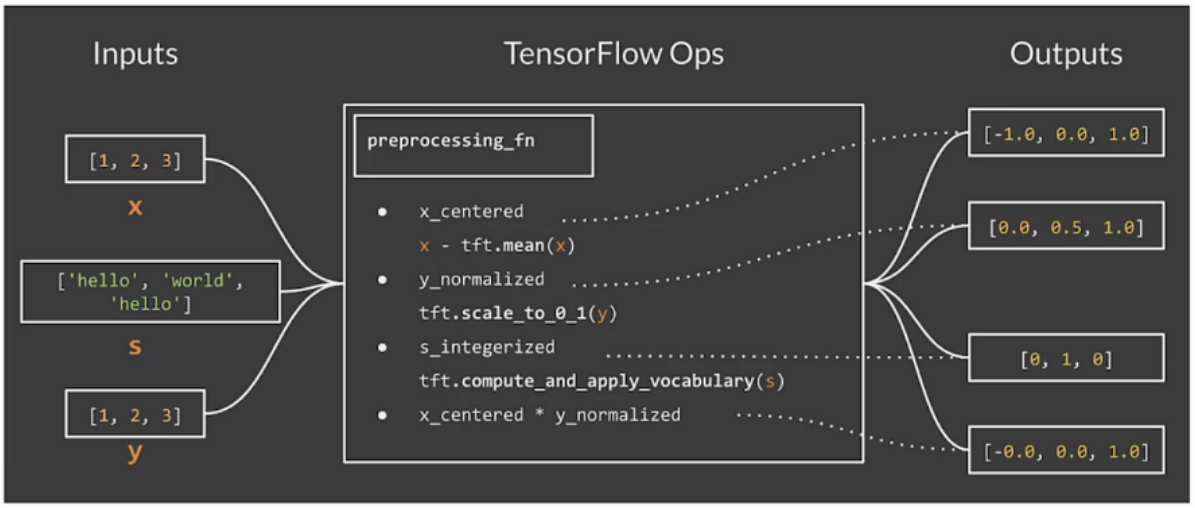
Preprocessing data (Transform)

```
def preprocessing_fn(inputs):
"""Preprocess input columns into transformed columns."""
    x, y, s = inputs['x'], inputs['y'], inputs['s']
    x_centered = x - tft.mean(x)
    y_normalized = tft.scale_to_0_1(y)
    s_integerized = tft.compute_and_apply_vocabulary(s)
    x_centered_times_y_normalized = (x_centered * y_normalized)
```

Preprocessing data (Transform)

```
    return {
      'x_centered': x_centered,
     'y_normalized': y_normalized,
     's_integerized': s_integerized,
     'x_centered_times_y_normalized': x_centered_times_y_normalized,
    }
```

Tensors in... tensors out



Running the pipeline

```
def main():
with tft_beam.Context(temp_dir=tempfile.mkdtemp()):
  transformed_dataset, transform_fn = (
      (raw_data, raw_data_metadata) | tft_beam.AnalyzeAndTransformDataset(
          preprocessing_fn))
```

Running the pipeline

```
  transformed_data, transformed_metadata = transformed_dataset



  print('\nRaw data:\n{}\n'.format(pprint.pformat(raw_data)))
  print('Transformed data:\n{}'.format(pprint.pformat(transformed_data)))

if __name__ == '__main__':
  main()
```

Before transforming with tf.Transform

```
    # Raw data:
    [{'s': 'hello', 'x': 1, 'y': 1},
     {'s': 'world', 'x': 2, 'y': 2},
     {'s': 'hello', 'x': 3, 'y': 3}]
```

After transforming with tf.Transform

```
  # After transform
  [{'s_integerized': 0,
    'x_centered': -1.0,
    'x_centered_times_y_normalized': -0.0,
    'y_normalized': 0.0},
   {'s_integerized': 1,
    'x_centered': 0.0,
    'x_centered_times_y_normalized': 0.0,
    'y_normalized': 0.5},
   {'s_integerized': 0,
    'x_centered': 1.0,
    'x_centered_times_y_normalized': 1.0,
    'y_normalized': 1.0}]
```

Key points

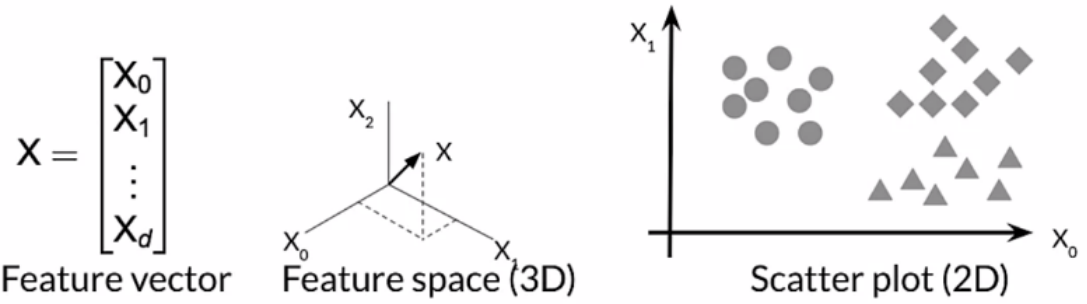
- tf.Transform allows the pre-processing of input data and creating features
- tf.Transform allows defining pre-processing pipelines and their execution using large-scale data processing frameworks
- In a TFX pipeline, the Transform component implements feature engineering using TensorFlow Transform

Feature Selection

Feature Spaces

Feature space

- N dimensional space defined by your N features
- Not including the target label



Let's see an example of house pricing.

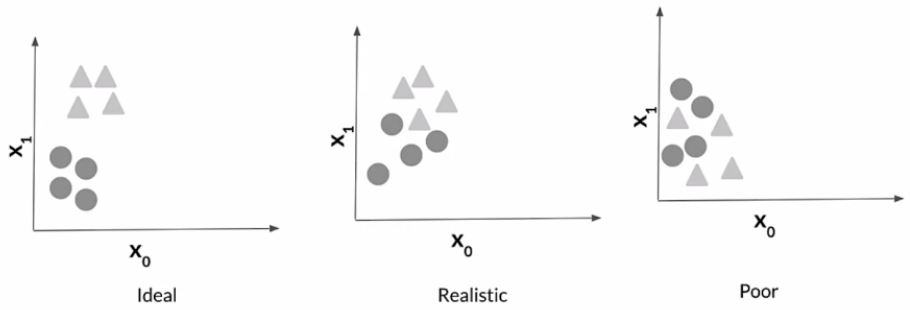
3D Feature Space

No. of Rooms X_0	Area X_1	Locality X_2	Price Y
5	1200 sq. ft	New York	$40,000
6	1800 sq. ft	Texas	$30,000

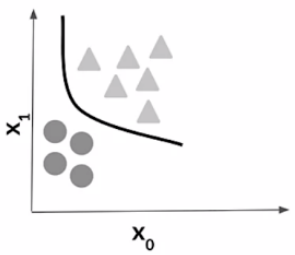
$Y = f(X_0, X_1, X_2)$

f is your ML model acting on feature space X_0, X_1, X_2

2D Feature space - Classification



Drawing decision boundary



Model learns decision boundary

Boundary used to classify data points

Feature space coverage

- Train/Eval datasets representative of the serving dataset
  - Same numerical ranges
  - Same classes
  - Similar characteristics for image data
  - Similar vocabulary, syntax, and semantics for NLP data

Ensure feature space coverage

- Data affected by: seasonality, trend, drift.
- Serving data: new values in features and labels.
- Continuous monitoring, key for success!

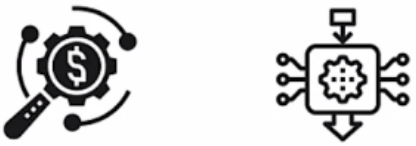
Feature Selection

Feature selection

- Identify features that best represent the relationship
- Remove features that don't influence the outcome
- Reduce the size of the feature space
- Reduce the resource requirements and model complexity

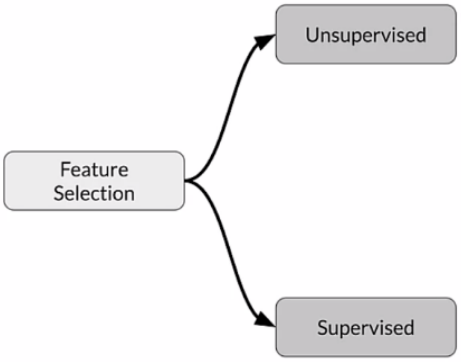
Why is feature selection needed?

Reduce storage and I/O requirements

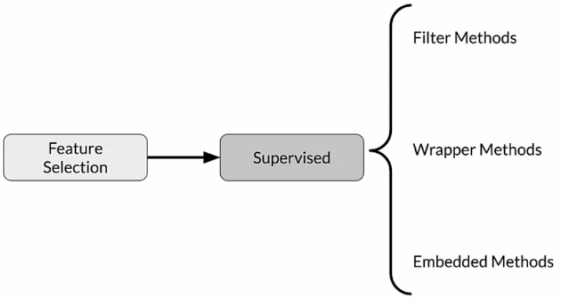


Minimize training and inference costs

Feature selection methods



Supervised methods



Unsupervised feature selection

1. **Unsupervised**

- Features-target variable relationship not considered
- Removes redundant features (correlation)

Supervised feature selection

2. Supervised

- Uses features-target variable relationship
- Selects those contributing the most

Practical example

Feature selection techniques on Breast Cancer Dataset (Diagnostic)

Predicting whether tumour is benign or malignant.

Feature list

id	diagnosis	radius-mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean
842302	M	17.99	10.38	122.8	1001.0	0.1184	0.2776
concavity_mean	concavepoints_mean	symmetry_mean	fractal_dimension_mean	radius_se	texture_se	perimeter_se	area_se
0.3001	0.1471	0.2419	0.07871	1.095	0.9053	8.589	153.4
smoothness_se	compactness_se	concavity_se	concavepoints_se	symmetry_se	fractal_dimension_se	radius-worst	texture_worst
0.0064	0.049	0.054	0.016	0.03	0.006	25.38	17.33
perimeter_worst	area_worst	smoothness_worst	compactness_worst	concavity_worst	concavepoints_worst	symmetry_worst	fractal_dimension_worst	Unnamed:32
184.6	2019.0	0.1622	0.6656	0.7119	0.2654	0.4601	0.1189	NaN

Irrelevant features

Performance evaluation

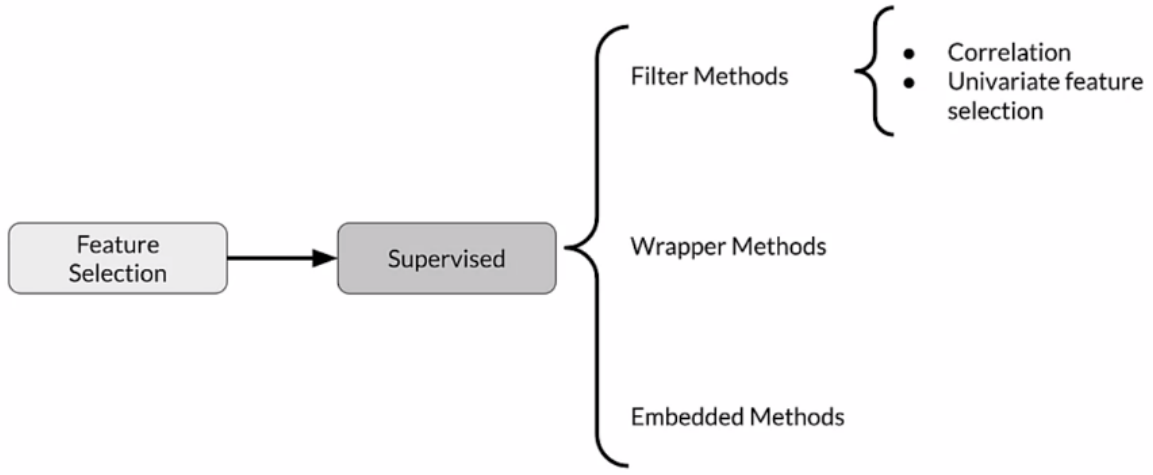
We train a **RandomForestClassifier** model in sklearn.ensemble on selected features

Metrics (`sklearn.metrics`):

Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.967262	0.964912	0.931818	0.97619	0.953488

Filter Methods

Filter methods

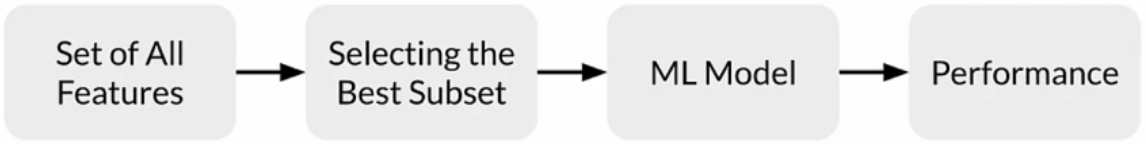


Filter methods

- Correlated features are usually redundant
  - Remove them!

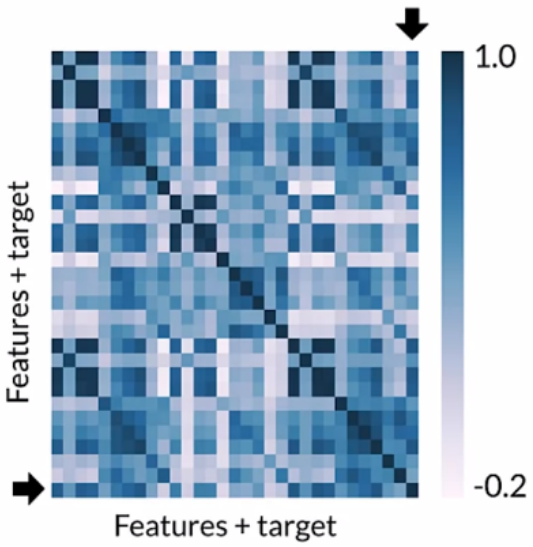
Popular filter methods:

- Pearson Correlation
  - Between features, and between the features and the label
- Univariate Feature Selection



Correlation matrix

- Shows how features are related:
  - To each other (Bad)
  - And with target variable (Good)
- Falls in the range [-1, 1]
  - 1 High positive correlation
  - -1 High negative correlation



Feature comparison statistical tests

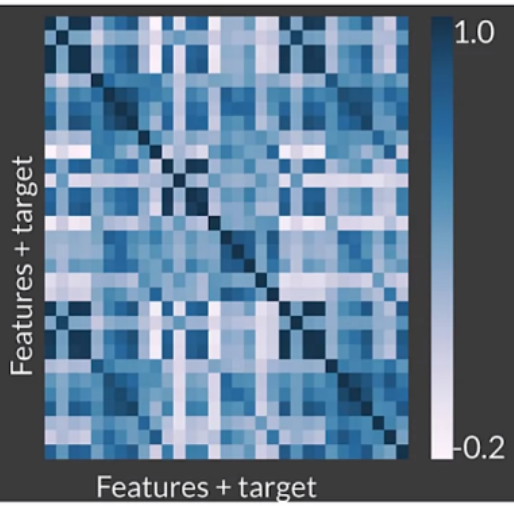
- Pearson's correlation: Linear relationships
- Kendall Tau Rank Correlation Coefficient: Monotonic relationships & small sample size
- Spearman's Rank Correlation Coefficient: Monotonic relationships

Other methods:

- Mutual information
- F-Test
- Chi-Squared test

Determine correlation

```
# Pearson's correlation by default
cor = df.corr()


plt.figure(figsize=(20,20))
# Seaborn
sns.heatmap(cor, annot=True, cmap=plt.cm.PuBu)
plt.show()
```



Selecting features

```
cor_target = abs(cor["diagnosis_int"])

# Selecting highly correlated features as potential features to eliminate
relevant_features = cor_target[cor_target>0.2]
```

Performance table

Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.967262	0.964912	0.931818	0.97619	0.953488
Correlation	**21**	**0.974206**	**0.973684**	**0.953488**	**0.97619**	**0.964706**

Univariate feature selection in SKLearn

SKLearn Univariate feature selection routines:

1. **`SelectKBest`**
2. `SelectPercentile`
3. `GenericUnivariateSelect`

Statistical tests available:

- Regression: `f_regression`, `mutual_info_regression`
- Classification: `chi2`, `f_classif`, `mutual_info_classif`

There is a bug in standardizing `X_test` here and in some of the next videos. It should be transformed using the parameters computed from `X_train` . You will see this correction in the ungraded lab at the end of this lesson.

SelectKBest implementation

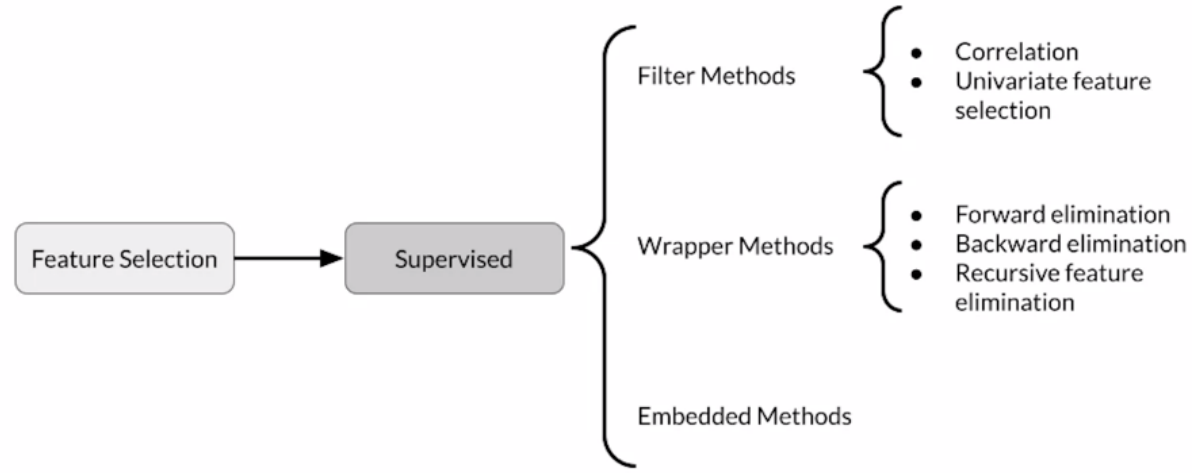
```
def univariate_selection():

  X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                          test_size = 0.2,stratify=Y, random_state = 123)

  X_train_scaled = StandardScaler().fit_transform(X_train)
  X_test_scaled = StandardScaler().fit_transform(X_test)

  min_max_scaler = MinMaxScaler()
  Scaled_X = min_max_scaler.fit_transform(X_train_scaled)
  selector = SelectKBest(chi2, k=20) # Use Chi-Squared test
  X_new = selector.fit_transform(Scaled_X, Y_train)
  feature_idx = selector.get_support()
  feature_names = df.drop("diagnosis_int",axis = 1 ).columns[feature_idx]
  return feature_names
```

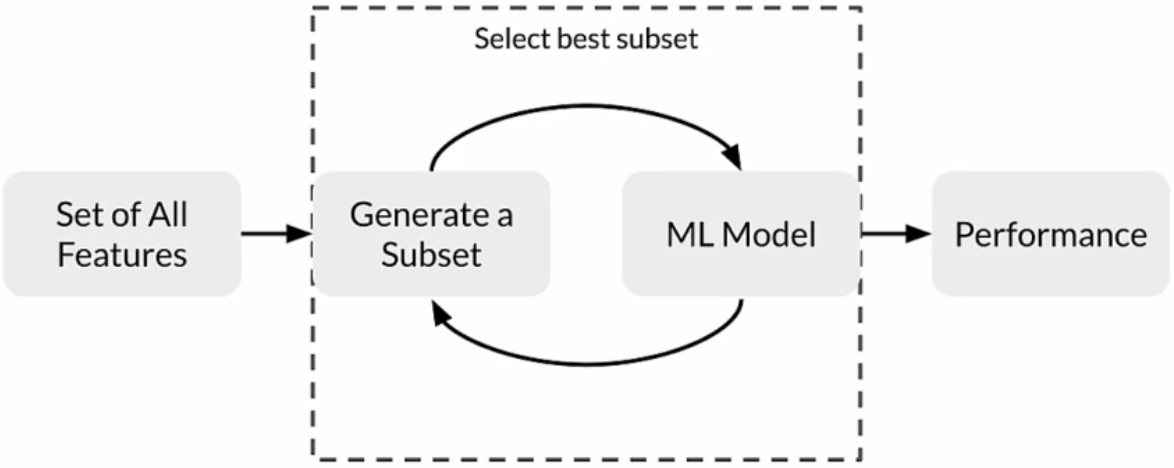
Performance table

Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.967262	0.964912	0.931818	0.97619	0.953488
Correlation	21	0.974206	0.973684	0.953488	0.97619	0.964706
Univariate (Chi^2)	**20**	**0.960317**	**0.95614**	**0.91111**	**0.97619**	**0.94252**

Wrapper Methods

Wrapper methods





Forward selection

1. Iterative, greedy method
2. Starts with 1 feature
3. Evaluate model performance when **adding** each of the additional features, one at a time
4. Add next feature that gives the best performance
5. Repeat until there is no improvement

Backward elimination

1. Start with all features
2. Evaluate model performance when **removing** each of the included features, one at a time
3. Remove next feature that gives the best performance
4. Repeat until there is no improvement

Recursive feature elimination (RFE)

1. Select a model to use for evaluating feature importance
2. Select the desired number of features
3. Fit the model
4. Rank features by importance
5. Discard least important features
6. Repeat until the desired number of features remains

Recursive feature elimination

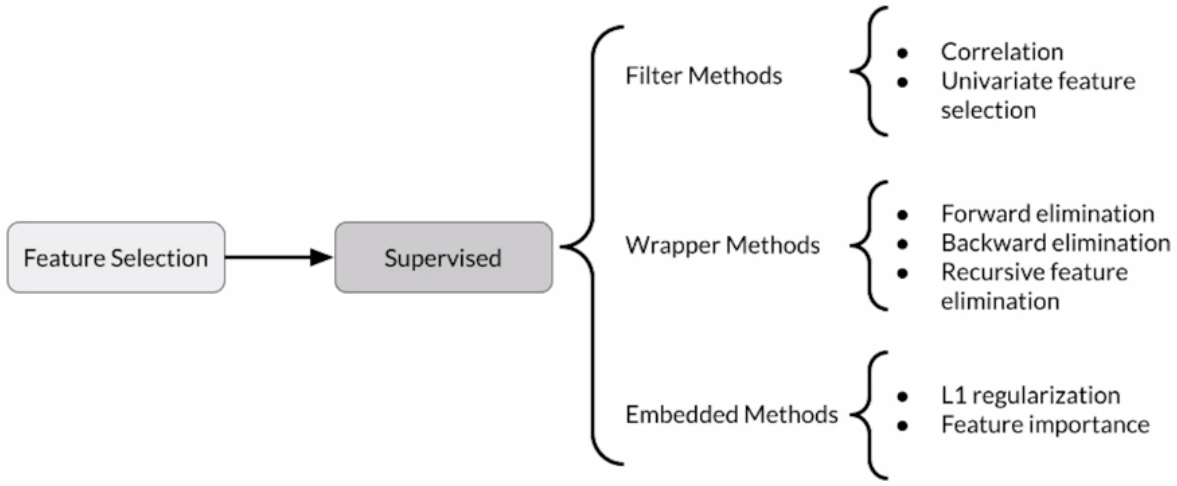
```
def run_rfe():

  X_train, X_test, y_train, y_test = train_test_split(X,Y, test_size = 0.2, random_state = 0)

  X_train_scaled = StandardScaler().fit_transform(X_train)
  X_test_scaled = StandardScaler().fit_transform(X_test)

  model = RandomForestClassifier(criterion='entropy', random_state=47)
  rfe = RFE(model, 20)
  rfe = rfe.fit(X_train_scaled, y_train)
  feature_names = df.drop("diagnosis_int",axis = 1 ).columns[rfe.get_support()]
  return feature_names

rfe_feature_names = run_rfe()

rfe_eval_df = evaluate_model_on_features(df[rfe_feature_names], Y)
rfe_eval_df.head()
```

Performance table

Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.96726	0.96491	0.931818	0.97619	0.953488
Correlation	21	0.97420	0.97368	0.9534883	0.97619	0.964705
Univariate (Chi^2)	20	0.96031	0.95614	0.91111	0.97619	0.94252
Recursive Feature Elimination	**20**	**0.97420**	**0.97368**	**0.953488**	**0.97619**	**0.964706**

Embedded Methods

Embedded methods



Feature importance

- Assigns scores for each feature in data
- Discard features scored lower by feature importance

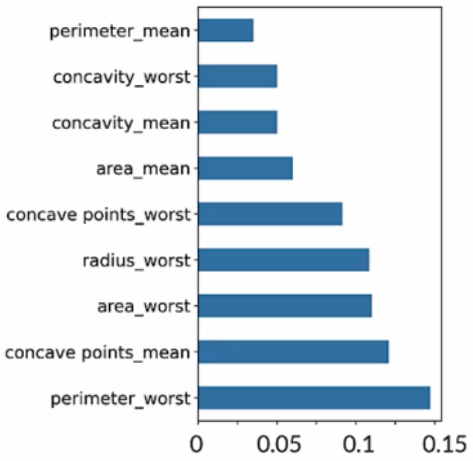
Feature importance with SKLearn

- Feature Importance class is in-built in Tree Based Models (eg., `RandomForestClassifier`)
- Feature importance is available as a property `feature_importances_`
- *We can then use* `SelectFromModel` *to select features from the trained model based on assigned feature importances.*

Extracting feature importance

```
def feature_importances_from_tree_based_model_():

  X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2,
                                                      stratify=Y, random_state = 123)
  model = RandomForestClassifier()
  model = model.fit(X_train,Y_train)

  feat_importances = pd.Series(model.feature_importances_, index=X.columns)
  feat_importances.nlargest(10).plot(kind='barh')
  plt.show()

  return model
```

Feature importance plot



Select features based on importance

```
def select_features_from_model(model):


  model = SelectFromModel(model, prefit=True, threshold=0.012)

  feature_idx = model.get_support()
  feature_names = df.drop("diagnosis_int",1 ).columns[feature_idx]
  return feature_names
```

Tying together and evaluation

```
# Calculate and plot feature importances
model = feature_importances_from_tree_based_model_()

# Select features based on feature importances
feature_imp_feature_names = select_features_from_model(model)
```

Performance table

Method	Feature Count	Accuracy	ROC	Precision	Recall	F1 Score
All Features	30	0.96726	0.964912	0.931818	0.9761900	0.953488
Correlation	21	0.97420	0.973684	0.953488	0.9761904	0.964705
Univariate Feature Selection	20	0.96031	0.95614	0.91111	0.97619	0.94252
Recursive Feature Elimination	20	0.9742	0.973684	0.953488	0.97619	0.964706
Feature Importance	14	0.96726	0.96491	0.931818	0.97619	0.953488

Embedded methods combine the best of both worlds, filter and wrapper methods:

- Faster than wrapper methods: Wrapper methods are based on the greedy algorithm and thus solutions are slow to compute.
- More efficient than filter methods: Filter methods suffer from inefficiencies as they need to look at all the possible feature subsets.

Week 2 Optional References

Week 2: Feature Engineering, Transformation and Selection

If you wish to dive more deeply into the topics covered this week, feel free to check out these optional references. You won't have to read these to complete this week's practice quizzes.

Mapping raw data into feature

Feature engineering techniques

Scaling

Facets

Embedding projector

Encoding features

TFX:

1. https://www.tensorflow.org/tfx/guide#tfx_pipelines
2. https://ai.googleblog.com/2017/02/preprocessing-for-machine-learning-with.html

Breast Cancer Dataset

Course #3 Machine Learning Modeling Pipelines in Production

Course #4 Deploying Machine Learning Models in Production